

# Viewer's Discretion: Host Security in Mobile Code Systems

Philip W. L. Fong\*  
School of Computing Science  
Simon Fraser University  
Burnaby, B.C., Canada V5A 1S6  
`pwfong@cs.sfu.ca`

November 25, 1998

SFU CMPT TR 1998-19  
`ftp://fas.sfu.ca/pub/cs/techreports/1998/`

## Abstract

Mobile code computation is a new paradigm for structuring distributed systems. Mobile programs migrate from remote sites to a host, and interact with the resources and facilities local to that host. This new mode of distributed computation promises great opportunities for electronic commerce, mobile computing, and information harvesting. There has been a general consensus that security is the key to the success of mobile code computation. In this paper, we survey the issues surrounding the protection of a host from potentially hostile mobile programs.

Decades of research in operating systems has provided significant experience and insight into the nature of system security. Before we propose any new security model for mobile code systems, it is wise to question why the existing protection mechanisms found in distributed operating systems do not fully address the security needs of mobile code systems. We propose three security challenges that are distinctive of the mobile code phenomenon, namely, the establishment of anonymous trust (establishing trust with programs from unfamiliar origin), layered protection (establishing fine-grained protection boundaries among mutually-distrusting parts of the same process), and implicit acquisition (coping with the implicit nature of mobile program acquisition).

We also survey various approaches to protection in existing mobile code systems. We classify protection approaches into four categories: discretion, verification, transformation, and arbitration. We evaluate each category by looking at how well they address the security needs of mobile code computation.

---

\*This research was funded in part by a scholarship and an operating grant from the Natural Science and Engineering Research Council of Canada.

# 1 Introduction

Mobile code computation is a new paradigm for structuring distributed systems[15]. Mobile programs migrate from remote sites to a host, and interact with the resources and facilities local to that host. This new mode of distributed computation promises great opportunities for electronic commerce, mobile computing, and information harvesting. There has been a general consensus that security is the key to the success of mobile code computation. In this paper, we survey the issues surrounding the protection of a host from potentially hostile mobile programs.

Decades of research in operating systems has provided significant experience and insight into the nature of system security. Before we propose any new security model for mobile code systems, it is wise to question why the existing protection mechanisms found in distributed operating systems do not fully address the security needs of mobile code systems. It is through this understanding that we can identify the distinctiveness of mobile code security. In this paper, we propose three security challenges that are distinctive of the phenomenon of mobile code computation, namely, the establishment of *anonymous trust* (establishing trust with programs from unfamiliar origin), *layered protection* (establishing protection boundaries among mutually-distrusting parts of the same process), and *implicit acquisition* (coping with the implicit nature of mobile code acquisition).

This paper also surveys the approaches to protection in existing mobile code systems. We classify protection approaches into four categories, *discretion*, *verification*, *transformation*, and *arbitration*. We evaluate each category by looking at how well they address the security needs of mobile code computation.

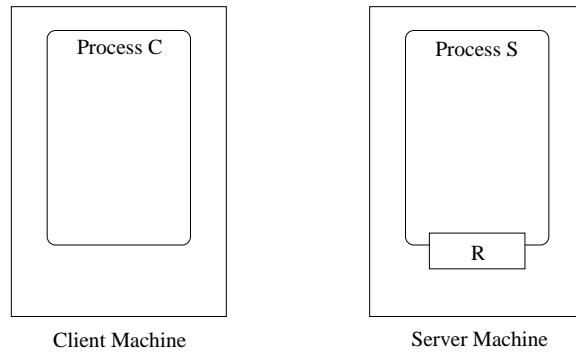
The paper begins with an introduction to mobile code systems (section 2). Section 3 describes the distinct security needs of mobile code computation. Section 4 surveys four major categories of protection approaches in existing mobile code systems, namely, discretion, verification, transformation, and arbitration. Research directions are summarized in section 5.

## 2 The Nature of Code Mobility

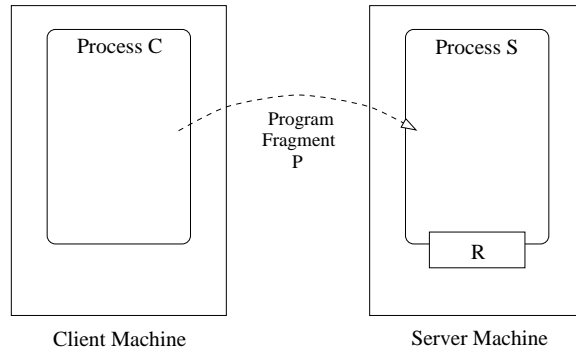
### 2.1 Code Mobility

Mobile code is a new way of structuring distributed software systems [15]. In general, one can always construct distributed systems by writing socket code using a client-server architecture. What is interesting about mobile code systems is the notion of *code mobility*: instead of simply passing data messages, communicating processes in mobile code systems exchange program code. Here, it is wise to distinguish between *strong mobility* and *weak mobility* [28]. Strong mobility is the mobility of *computation*: an entire thread (or a group of threads), including both its code and its state (e.g. binding environment, stack, etc) are transported. Weak mobility describes the motion of code only. For instance, various forms of active contents (e.g. Java applets [50]) are examples of weak mobility.

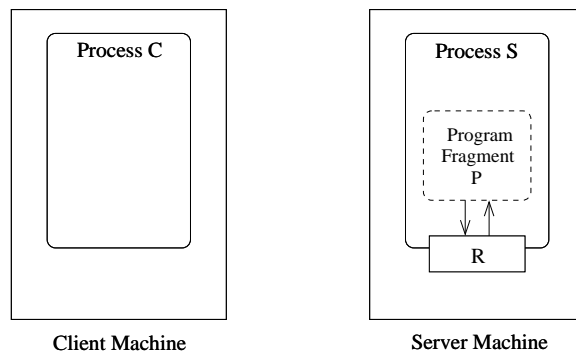
**Weak mobility.** A weak mobility system has a typical architecture as depicted in figure 1. A client process  $C$  and a server process  $S$  run in two different machines.



(a) Before



(b) Migration



(c) After

Figure 1: Weak Mobility

The server process  $S$  has access to some resource  $R$  in the server machine (Fig 1(a)). Resource  $R$  could be a data structure, some service routines, or simply computer cycles. Attempting to access resource  $R$  in the server machine, the client process  $C$  sends a program fragment  $P$  to the server process  $S$  (Fig 1(b)). Upon arrival, code fragment  $P$  is dynamically linked into process  $S$ .  $S$  then invokes the code in  $P$  (Fig 1(c)), enabling  $P$  to access resource  $R$  on behalf of process  $C$ . The result of accessing  $R$  is communicated back to  $C$  if necessary. To make the above abstract description more concrete, think of  $S$  being your web browser process, with access to your local windowing system and terminal display (resource  $R$ ), and think of  $C$  being an `httpd` process. The code fragment  $P$  could be a Java applet.

Carzaniga *et al* [15] identify two variants of the above architecture:

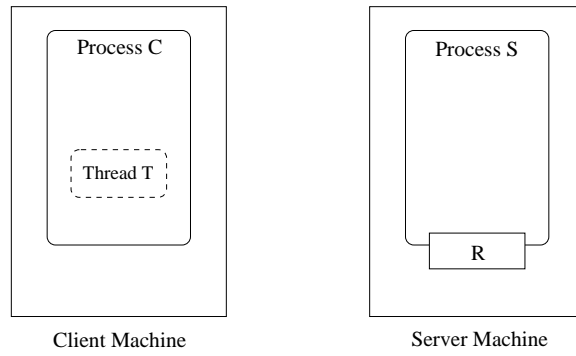
**Code-on-Demand:** The server process  $S$  initiates communication. It requests  $C$  to send it  $P$ . Various forms of active contents, including Java applets [50], ActiveX control [112], and JavaScript [80] are commercial examples of code-on-demand.

**Remote Evaluation:** The client process  $C$  initiates the communication. It sends  $P$  to  $S$ , requesting  $S$  to evaluate  $P$  on its behalf. Stamos and Gifford [108] were the first to envision this architecture. One could view Java servlets [18], MIME Tcl extension [13], and active packets [54] as examples of remote evaluation.

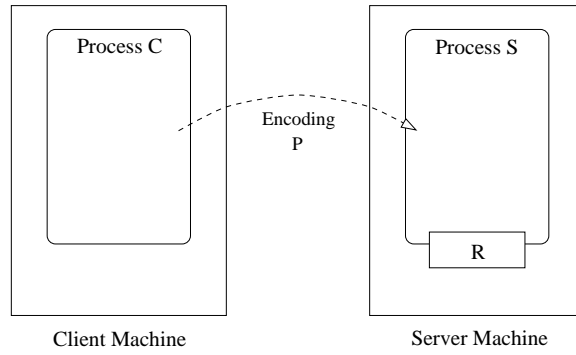
**Strong mobility.** Strong mobility is commonly known as mobile agents [15]. A typical mobile agent architecture is depicted in figure 2. A group of coordinating threads  $T$  is originally running in a process  $C$  on the client machine, and accessing the resources therein (Fig 2(a)). At certain point, thread group  $T$  attempts to access resource  $R$  that is owned by a process  $S$  running on a remote machine. To do so, thread group  $T$  requests its parent process  $C$  to transport it to remote process  $S$ . Process  $C$  suspends  $T$ , and then marshals  $T$  into an encoding  $P$  that encapsulates both the code  $T$  is executing and the dynamic state of  $T$ . The encoding  $P$  is then sent off to process  $S$  (Fig 2(b)). Upon arrival, the encoding  $P$  is dynamically linked into process  $S$ , and the execution of  $T$  is resumed, thereby allowing  $T$  to interact with resource  $R$  (Fig 2(c)).  $T$  remains in  $S$  until there is a need to access remote resource again.

**Generalization.** Let us ponder again on the two architectures laid out in figure 1 and 2. The whole point of exchanging code in both cases is for the an external party to access the remote resource  $R$  located in the server machine. In every mobile code system, there is a computing platform, called a *host*, that offers external access to a selected set of computing resources that it owns. The resources are managed by a process called a *computing environment (CE)*, which defines a controlled interface between the resources and external parties who want to access the resources. In our example, process  $S$  fills this role of a computing environment.

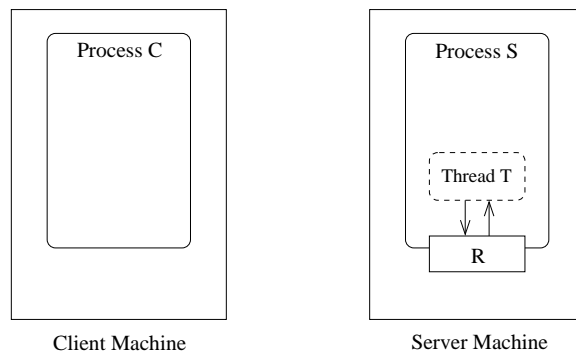
When an external party attempts to access the facilities managed by the computing environment, it sends off a *mobile code unit* which corresponds to either a code fragment (as in the case of weak mobility) or an encoding of some threads (as in the case of strong mobility). The program code in the mobile code units will then be dynamically linked into the computing environment process. One or more threads will be started to carry out the computation prescribed by the downloaded mobile code unit, that is, to access the resources of interest on behalf of the external party. These threads might in turn



(a) Before



(b) Migration



(c) After

Figure 2: Strong Mobility

start some other threads. Together these threads constitute a recognizable application program that carry out a single job. We call such a group of threads an *execution unit*. An execution unit is a direct analogue of a process in a traditional operating system, just as the computing environment is an analogue of the operating system itself. No execution unit is allowed to access host resources unless by conforming to the protocol imposed by a computing environment. Notice that a host may serve multiple computing environments, while each computing environment may in turn host one or more execution units.

We use Java as an example to illustrate the above ideas. The Java virtual machine (JVM) is the computing environment of Java. As an interpreter, it creates a *sandbox* in which interaction between Java threads and the underlying operating system can be controlled. Multiple JVM can coexist at a single host. Migration occurs when the a Java classloader retrieves mobile code units, called Java classfiles, from, say, a remote `httpd` process. The classfiles are verified and then dynamically linked into the JVM. Finally, the JVM spawns off an applet thread that invokes the linked code. An applet is therefore an execution unit in Java.

## 2.2 Motivation of Code Mobility

Motivation for adopting the mobile code paradigm has been surveyed in great detail in the literature [61, 21, 22, 15]. The following list illustrates several representative ones:

1. **Real-time interaction with remote resources:** Most computing resources, like databases, file systems, or even physical displays, are not transportable. If such resources are located in a remote site, then computation that requires *real-time interaction* with the resources has to happen where the resources reside. Code mobility allows one to prescribe the location of computation, so as to make real-time interaction possible. For example, active contents like Java applets prescribe interactive presentation that is to be rendered on the browser side.
2. **Reduction of Communication Traffic:** Mobile computers (e.g. hand-held computers or intelligent mobile phones) usually interact with servers through unreliable, low-bandwidth, high-latency, high-cost networks. Mobile agents becomes an attractive alternative because network traffic can be reduced by migrating the client program to the server side, thus avoiding the potential cross-network communication bottlenecks.
3. **Customization and extension of server capabilities:** Valuable hardware resources are usually managed by server software (e.g. an operating system). Such server software usually define access policies that are extremely general, and tend to ignore the specific needs of individual clients. Recently, various proposals have been made to allow application-specific extension code to be downloaded dynamically into a server software, so as to customize its access policies to meet the specific need of a client software. Typical examples include extensible operating systems [11, 102, 37], active network [111, 20], active disk [4], and many others.
4. **Avoiding distribution of state:** In traditional client-server applications, the state of computation is distributed among the servers and the clients. As a consequence, it is difficult to maintain consistency of the distributed states, and to articulate the correctness of the computation. Mobile agents localize computation

states in a single process. They offer a better abstraction that makes the crafting of distributed software a more manageable task.

Chess *et al* [22] fairly pointed out that any application that can be crafted under the mobile code paradigm can also be structured as a client-server application. However, as we have seen above, mobile code systems offer many engineering advantages that its client-server counterpart lacks. Recently, Carzaniga *et al* [15] propose an abstract model for evaluating the potential benefit of adopting various mobile code paradigms. The result suggests that the advantages pay off only for certain kinds of application domains.

## 2.3 Code Mobility, Dynamic Linking, and Binding Environment

The essence of code mobility is the ability to transport some resource-accessing mobile code units from one host to another, and then executing those code units on the resource-bearing host. In order for the resource-accessing mobile code unit to do so, it must contain some free variables that refer to the resources it is interested in. Therefore, upon arrival, all mobile code units must go through a phase of integrating into the computing environment, and that integration usually takes the form of dynamic linking, that is, linking the free variables in the mobile code units to the actual resources owned by the computing environment.

Viewing from a programming language perspective, mobile code units are basically code that contains free variables. A computing environment is in fact a *binding environment* that defines bindings for the free variables in mobile code units. Code migration, then, is the dynamic specification of the binding environment in which an open program is to be executed. This view, first advocated by Queinnec and De Roure [90], can be illustrated in the following way.

The signature of a typical metacircular Scheme interpreter<sup>1</sup> [3] has the following form:

```
(eval form env)
```

Given a Scheme form (an open program) and a binding environment, the `eval` interpreter computes the value of `form`, using the bindings in `env` whenever free variables in `form` are to be resolved. If we see a computing environment as a binding environment that supplies bindings for the free variables occurring in a mobile code unit, then code mobility can be viewed as the explicit identification of the binding environment in which a form is to be evaluated. It is this dynamic linking aspect that makes computation “occur at another site”. In this manner, remote evaluation can be understood as the evaluation of a form in a remote environment:

```
(eval form network-locator-for-environment)
```

Here, `network-locator-for-environment` names the binding environment in which `form` will be evaluated. Code-on-demand can be understood similarly:

```
(eval network-locator-for-program local-environment)
```

---

<sup>1</sup>A metacircular Scheme interpreter is a Scheme interpreter written in the Scheme language itself.

Here, `network-locator-for-program` names the remote program that is to be brought in for execution.

Mobile agents are not easily understood in such a framework. In the studying of code sharing via the internet, Queinnec and De Roure [90] propose the following special form that captures various forms of code mobility:

```
(import (v1 v2 ...) env form)
```

The `import` special form composes an environment, and then evaluates `form` in this environment. The target environment is composed by taking the current environment (the lexical environment `import` is in) and then overshadowing the definition of variables `v1`, `v2`, ... by those in the first-class environment `env`. If `env` is a remote host, then the target environment will be composed of both the bindings in the remote host and the current state of computation. Strong mobility is thus given a very clean model.

Realizing that dynamic linking and dynamic specification of binding environment are the underlying reality of mobile code computing is crucial for the following reason: there is a very subtle analogy between a protection domain and a name space (a binding environment).

## 2.4 Viewer's Discretion: The Security Challenge

PostScript files [109] could be seen as a form of mobile programs [116]. As a stack-based, page-description language, it offers a very compact description of documents. When PostScript documents are sent to printers, the documents are interpreted to generate printout. This setup off-loads some of the document rendering burden from the printing hosts to the printers. Because of its simplicity, one does not usually expect this setup to go wrong.

As PostScript gradually gained its popularity as a portable document description language, it became a standard medium for document distribution on the internet. Recognizing it as a MIME type, web browsers automatically spawn off a viewer process as a Postscript file is downloaded. In 1995, the CERT coordination center discovered that some older versions of postscript viewers allow Postscript programs to access the local file system in an unsafe manner [16]. This presents a potentially serious threat to the web browsing system, for the vulnerability can be abused by malevolent programmers "intentionally embedding commands within an otherwise harmless image so that when displaying that image the PostScript viewer may perform malicious file creations or deletions." [16]

With the emergence of *rogue applets* [62, 68] and other forms of malicious active contents, users of mobile code systems are now more aware of the security threats associated with mobile code computation. A rogue mobile code unit may overwrite valuable data on local disks, covertly transmit private information to another party, hang the hosting browsers, or masquerade as another trusted application.

As the owner of a host, security is an issue for us because the code to be executed by our host does not belong to us. We never write them, nor do we know a lot about them, and sometimes we do not even know who sent them to our host. Allowing such programs to be executed without control is like inviting strangers into your house — you are lucky if nothing bad happens.

A naive response would be to turn off all mobile code capabilities, or to completely



avoid visiting web sites with active contents. However, active contents are becoming increasingly popular. Businesses are conducted around them. This avoidance mentality only cripples us. The question then is not one of avoiding download, but of protecting ourselves from the downloaded execution units that run wild.

Security is also the most challenging aspect in the crafting of mobile code hosting environment. For one, it is difficult to articulate exactly what we mean by secure mobile code computing. In the next section, we will delineate the distinct security needs of mobile code systems.

### 3 The Distinct Security Requirements of Mobile Code Systems

There are two classes of security issues in mobile code computing:

- *Host security* is concerned with the protection of a host from untrusted mobile programs, and with the avoidance of mutual interference among execution units.
- *Application security* is concerned with the assurance of correctness and confidentiality for the computation that is delegated to a remote host. When an untrusted host carries out a computation on behalf of a client, the host may maliciously corrupt or expose the internal state of the client's execution units. Application security goes by other names like *code security* [119].

This paper is mainly devoted to the exploration of issues in host security. Readers interested in application security may consult the paper by Farmer *et al* [39] for an inspiring discussion.

Viewing from the perspective of host security, mobile code systems share many similarities with an operating system. In particular, security issues arise in both kind of systems when they attempt to address the need of *multiprogramming*, and specifically, the need for safe resource sharing. A multiprogramming system allows multiple execution units to coexist on a host. It is desirable to ensure that no single execution unit has total monopoly of the host's resources. As a result, individual execution units must be guarded from interfering with each other, and from undesired exploitation of the host's resources and facilities. To realize this, various protection mechanisms are built into an operating systems. Silberschatz and Galvin [103, p 431] understand the entire business of protection as one of *providing safe sharing of name spaces*:

*Protection was originally conceived as an adjunct to multiprogramming operating systems, so that untrustworthy users might safely share a common logical name space, such as a directory of files, or share a common physical name space, such as memory.*

Given the common security goal of safe resource sharing, it is then important to question why the existing protection mechanisms in operating systems cannot be directly transplanted to mobile code systems to address its similar security needs. In other words, we need to articulate in clear and precise terms an answer to the following question:

**Question of Distinction:** What makes the security needs of mobile code computing different from those of traditional multiprogramming operating systems?

This section attempts to answer the above question.

In section 3.1–3.3, we will give a brief survey of the security requirements and protection mechanisms in traditional multiprogramming systems. Details can be found in standard textbooks on operating systems [103], computer architecture [87], distributed systems [26], and computer security [88]. Readers already familiar with these topics may skip the mentioned sections. In section 3.4, we propose three distinctive challenges in mobile code security. Then in section 3.5 we articulate why software-based solutions to such challenges should be sought for.

### 3.1 Ingredients of Host Security

There are three aspects to the security of a computing system:

**Integrity:** System resources should be protected from unauthorized modification, deletion, or other means of tampering.

**Confidentiality:** Sensitive information should be protected from leaking through unauthorized channel. Confidentiality goes by other names like *privacy* or *secrecy*.

**Availability:** The computing system should be protected from interference that affects its normal operation and availability of service.

In order to establish and evaluate the security of a computing system, one has to refine the above criteria, and lay out exactly what the security requirements are in concrete terms. A *policy* is a well-defined, consistent, and implementable statement of the security we expect the system to enforce. It is an administrative decision about what should constitute a breach of security. Users of mobile code system should watch out for the following *threats*:

**Corruption.** Execution units may modify or erase important data. Other may tamper with the internal state of the system, rendering the system state incoherent. Typical examples of such attacks include various forms of computer virus that cause damages to host's data. Such attacks compromise the integrity of the system.

**Leakage.** Execution units may actively release sensitive information to an outside party. Other may engage in data processing activities from which malicious third parties can infer information that is supposed to be classified (i.e. covert channels). For example, an unguarded Java applet may access personal financial records on an unsuspecting PC's hard drive, and then send the information back to the attacker via a socket connection. Such attacks are direct violation of the system's confidentiality.

**Denial of service.** Execution units may monopolize shared resources like the terminal screen (creating a window so huge that it covers everything on your terminal screen), CPU time (raising its own priority above all other threads), threading services (killing all threads other than itself), etc. Such attacks destroy the availability of the system.

**Masquerading.** Execution units may masquerade as another one by faking the user interface of the latter, thus fooling the users into entrusting them with critical resources (e.g. stealing CPU time for factoring an integer) and data (e.g. asking users for their passwords). Execution units may also pretend to originate from a trusted origin (e.g. various spoofing attacks). An execution units may even fool the type system by appearing to be of another type (e.g. type confusion in Java), thus gaining access to the internal state of the system. Masquerading is a very subtle form of attack that could potentially lead to the compromising of integrity, confidentiality, and availability.

For more details of the above threats in a concrete mobile code system Java, consult other surveys [62, 68, 40, 86].

## 3.2 Protection Mechanisms and Their Evaluation Criteria

Protection mechanisms are technologies built into the computing environment for the sake of enforcing security policies. Protection is based on the notion of *separation* [96]. Separation can be *physical* (allocating physically distinct resources to competing parties), *temporal* (scheduling competing processes to execute at a different time), *logical* (creating logical barrier to avoid interference), or *cryptographic* (encrypting sensitive information).

### 3.2.1 Design Principles of a Secure System

Saltzer and Schroeder [97, 98] list the following principles for the design of secure protection mechanisms:

1. **Economy of mechanisms.** The design of the protection mechanism should be small and simple. A small and simple mechanism can be carefully analyzed and validated.
2. **Fail-safe default.** The default condition should be denial of access. The designer of a protection mechanism should determine what is accessible instead of when access is denied.
3. **Complete mediation.** The protection mechanism should be designed so that all possible access to system resources are covered. In a system that will be used continuously, and in which access rights may be revoked, every access attempt should be checked.
4. **Open design.** The security of the protected system should not depend on keeping the design of the protection mechanism secret.
5. **Separation of privilege.** Access to an object should depend on more than one condition. In this way, complete security breach will not occur even when one protection system is defeated.
6. **Least privilege.** Execution units should be granted the bare minimum amount of privileges necessary to complete the job.

7. **Psychological acceptability.** If the users feel that protecting their system resources is too much work, they will not use it. The human interface should be designed for naturalness, ease of use, and simplicity, so that users will routinely and automatically apply the protection mechanisms.

### 3.2.2 What Makes a Protection Mechanism Reasonable

Maximizing security is an apparent goal. Yet absolute security can trivially be achieved by not allowing any foreign code to be executed in a host, an alternative we do not want to live with. Therefore, our goal is really to maintain security while retaining the usefulness of mobile programs. In particular, we want to achieve the following objectives:

1. **Fine granularity of control.** An execution unit that cannot do anything can do no harm, but a execution unit must be granted enough privileges in order to be useful. A competent protection mechanism should flexibly express the precise needs of an execution unit. Granting too many privileges will compromise the principle of least privilege (section 3.2.1), while granting too few impairs the execution unit.
2. **Minimal interference with normal operation.** In order not to introduce performance degradation to a mobile program, the computation environment should avoid unnecessary interference with downloading, linking, and executing execution units.

In summary, the impact of protection should as small as possible.

## 3.3 Traditional Protection Mechanisms

Two relevant protection mechanisms in traditional operating systems are *memory protection* and *access control*<sup>2</sup>.

### 3.3.1 Memory Protection

*Memory protection* guarantees that (1) execution units do not interfere with the states of each other and do not corrupt the global state of the host; (2) control can flow outside of the execution unit only through some well-defined interface provided by the computing environment. In short, the goal of memory protection is to avoid the malfunctioning of one execution unit from contaminating other execution units or even the host itself. Memory protection also goes by other names in the mobile code literature, namely, *low level security* [126] or simply *safety* [59].

In traditional operating systems, memory protection is achieved by three mechanisms in combination:

---

<sup>2</sup>In fact, competent operating systems offer other kinds of protection mechanisms, including CPU protection, instruction protection, I/O protection, etc. In a mobile code system, execution units usually execute inside a user process, and thus are already protected from monopolizing the CPU(s), executing privileged instructions, and directly accessing any I/O channels. Therefore, we will not touch on these issues in the remainder.

1. Processes are placed in separate address spaces. At run-time, every address reference, be it a data reference or a control transfer, is checked by the hardware to see if the referenced location belongs to the address space of the running process. If not, the memory fault will be trapped by the operating system.
2. The CPU offers two mode of execution, namely, the kernel mode and the user mode. User processes are always executed in the user mode. Instructions that set the boundary of address space are protected, and can only be executed in the kernel mode. User processes are then protected from redefining the boundary of their address spaces.
3. Control flowing outside of the address space must pass through special interface of the operating system kernel. This is usually achieved by providing a set of pre-defined system calls accessible by a special TRAP instruction. When a system call is invoked, the CPU switches to kernel mode, and control is transferred to the operating system, which in turn processes the system call on behalf of the user process.

Instead of a simple dichotomy of kernel and user modes, some operating systems provide multiple, concentric *rings* of security levels. Code running in the more trusted rings may access data and code in the less trusted ring, but not vice versa. Less trusted code may transfer control to the more trusted code via special entry points called *gates*. Such arrangement requires special hardware and operating system support.

### 3.3.2 Access Control

In traditional operating systems, resources of the host are modeled as *objects*, while user processes are modeled as *subjects*. A subject can perform *operations* on an object. The permission to perform a certain operation on an object is said to be an *access right*. Security policies are expressed as an assignment of rights to subjects. A *protection domain* is a collection of access rights. A user process acquires its access rights by being associated to a protection domain.

Access rights in a multiprogramming system can be expressed as an *access matrix*. Every row in the matrix represents a protection domain, while every column represents an object. By labeling an entry in the matrix with access operations, we effectively define what rights are given to a protection domain (row) for the accessing of an object (column). In traditional operating systems, an access matrix is usually implemented in one of two ways. The first approach is the *access control list*. Associated with every system resource is a list of  $\langle \text{subject}, \text{right} \rangle$  pairs. Whenever a resource is accessed, its associated list is checked to see if the accessing subject is on the list and is granted appropriate access right. A second approach is that of *capability*. A capability is an unforgeable pointer to a system resource. By acquiring the pointer, a subject is then granted the right to access the object. In a sense, capability controls access through visibility — if a system resource is not even visible to a process, then there is no way the process can access it.

In traditional operating systems, access control presumes two related mechanisms: *authentication* and *authorization*. Authentication is the process of establishing the identity of a user. Authorization is the process of translating the result of authentication into decisions of what access right is granted to authenticated users. Under

such system, whether a user has right to perform an operation is largely dependent on his/her identity, or, more precisely, on the operating system's knowledge about him/her.

### 3.4 Distinctiveness of Mobile Code Security

What then distinguishes mobile code computing from multiprogramming in traditional operating systems? Why cannot the traditional protection mechanisms satisfy the security needs of mobile code computing? We believe that the mobile code phenomenon is distinctive in the following way:

**Distinctiveness of Code Mobility:** Subject only to *time-bounded, automatable checking*, code originating from *any arbitrary source* may be executed in an environment that *exposes access to shared resources*.

Here, the three key phrases are “time-bounded, automatable checking”, “any arbitrary source”, and “exposes access to shared resources”. In the following, we will explain these three distinctive aspects of mobile code computing.

#### 3.4.1 Anonymous Trust

Traditional security models are based on trusted identities. A user is a known party. Based on one's trust with the user's identity, and based on notions like resource ownership, one then authorizes access to system resources. A direct translation of this idea to mobile code security is to attach to every mobile code unit a digital signature that signifies its origin. In this view, the source of the mobile code unit is treated as a user in a traditional operating systems, and authorization of access rights is based on the familiarity of the code unit's origin. This approach works well when the mobile program is developed by a well-known brand name, or when it is sent from a trusted source. Yet, the approach breaks down when the foreign code is written by an unknown author and sent from an obscure origin. The very spirit of WWW computing is that any party can freely share information or active content with others who have access to the internet. It is conceivable that in the future more and more of the code we use is going to be developed and distributed by parties unknown to us. Security that is based solely on identity cannot account for such phenomenon. Under such an approach, the only completely safe practice will be to reject all untrusted code. However, such a restraining policy degrades the granularity of control. This difficulty was first articulated by Ousterhout *et al* [84], and then found its full expression in a paper of Chess [23].

The above discussion illustrates a fundamental challenge in mobile code security:

**Anonymous Trust:** How do we establish trust for a mobile code unit sent from an unknown origin and written by an unknown party?

Given the above, security engineers should not rely solely on identity or origin information to authorize access. A competent security infrastructure should accept trustworthy mobile programs even if they are anonymous. Therefore, we accept as an axiom that mobile programs from *any* origin will be downloaded into a computing environment:

**Axiom of Anonymity:** Code from any arbitrary origin will be downloaded.

Another way to look at the issue is that, with the emergence of the web, *the model of software distribution has changed*. No longer can we establish trust solely by means of brand names and well known vendors, for everyone may now distribute software. We need to put tools and techniques into the hands of this new breed of anonymous programmers so that they can construct software that *inspires* trust in hosts. Devanbu *et al* call such an endeavor *trusted software engineering* [35].

### 3.4.2 Layered Protection

Another fundamental aspect of mobile code computing is that a mobile code system defines a complete multiprogramming environment on top of the existing operating system. The mobile code computing environment may define its own computing model, maintain its own resources, provide its own set of services, and hence define its own security model. Consequently, it is usually not realistic to simply treat an execution unit as just another process in the operating system, running in just another protection domain. Our desire for platform independence further discourages us from adhering to the security model of a particular operating system.

In addition, as one of the users in the underlying platform, a mobile code computing environment may *expose* some of the operating system resources to the visiting execution units. Thus, it is impossible to construct a mobile code security model which is completely orthogonal to the underlying operating system's security model. The mobile code security model must honor the security constraints imposed by the operating system.

A typical arrangement in existing mobile code systems is to have the computing environment running as an operating system process, while treating execution units as secondary threads executing within that process. For example, a web browser is a single process containing a Java virtual machine, in which Java applets are run. A Java web server is also a single process. Servlets run within the server process as secondary threads, thus avoiding costly CGI-based solutions. Since a process defines both the boundary for memory protection and the protection domain for access control, execution units are protected from exploiting resources outside of the computing environment.

Protection within a computing environment is less trivial. Mobile code units are linked into the computing environment process. Sharing the same address space and protection domain, the execution units and the computing environment process are indistinguishable from the point of view of the operating system, which must rely on process boundary to enforce protection. We then have an interesting situation in which code fragments within a single application process do not trust each other<sup>3</sup>. As

---

<sup>3</sup>It is revealing to read a passage in a standard operating systems text [103, p 112]:

*However, unlike processes, threads are not independent of one another. Because all threads can access every address in the task, a thread can read or write over any other thread's stacks. This [multi-threading] structure does not provide protection between threads. Such protection, however, should not be necessary. Whereas processes may originate from different users, and may be hostile to one another, only a single user can own an individual task with multiple threads. The threads, in this case, probably would be designed to assist one another, and therefore would not require mutual protection.*

The introduction of mobile code computing environment definitely falsifies the above assumption.

a result, the computing environment process has to enforce memory protection and access control<sup>4</sup> within the process itself.

The security model of the operating system and the security model of the mobile code system then form a parent-child relationship. Providing mechanisms for defining child protection domains inside a single process becomes the second fundamental challenge to mobile code security:

**Layered Protection:** How do we implement protection among mutually-distrusting code and execution units coexisting in a single application process?

An alternative articulation of this problem is what Rees [91, p 16] called the *safe invocation problem*:

*When a program is invoked, it inherits all of the privileges of the invoker. The assumption is that every program that a user runs is absolutely trustworthy. ... It is remarkable that we get by without secure cooperation. We do so only because people who use programs place such a high level of trust in the people who write those programs.*

What Rees refers to by secure cooperation is the ability to invoke an unfamiliar routine without granting it all the invoker's rights. To achieve this, one needs to be able to hierarchically construct a child protection domain for executing untrusted code within the invoker's process.

Layered protection is a characteristic feature in single-address-space operating systems like OPAL [19] and extensible operating systems like SPIN [11], VINO [102], and Exokernel [37]. In such operating systems, untrusted code may be (dynamically) introduced into a privileged protection domain (e.g. the kernel). One wants to avoid the untrusted code units from exploiting the resources in that domain. In the following discussion, we will refer to some of such operating system literature that offers better understanding of layered protection in mobile code systems.

### 3.4.3 Implicit Acquisition

Code mobility defines a new model of *software acquisition*. In the past, acquisition of commercial-off-the-shelf software package involves a slow, manual, and explicit process. Alternatives are reviewed and tested. Impact analysis is conducted. Deployment is planned and staged. System administrators know exactly what packages are installed on the system. Potential impact is announced to users.

Software acquisition is completely different in a mobile code system. As De Paoli *et al* [86] put it:

*Conventional computing paradigms assume that programs are installed and configured once on any and every machine and that these programs only exchange data. This means that a user can make all possible checks over a new program before running it. This assumption, however, is no longer valid for open and mobile environments, such as Java and the Web.*

---

<sup>4</sup>Again, one could point out the need to avoid the monopolization of the process's time slice by a secondary task. This is easily achieved if we have a competent scheduling mechanism. Therefore we will leave the issue here. For those interested in having more control on Java's scheduling mechanism, consult [63, section 8.3].



Mobile code unit may arrive even without the knowledge of the users. Simply by browsing a web page will invoke the installation of active content. Acquisition is therefore implicit. It is actually a design goal (minimizing interference) that the entire acquisition process be invisible to the users (see section 3.2.2, on Minimal interference with normal operation). In such a process, only *automatable checks* are allowed, be it signature checking, program analysis, type-checking, etc. All such checking should take only a *limited* time to complete. It is this time constraint that makes code mobility *a completely new model for software acquisition*. The traditional acquisition process establishes trust gradually. Yet, with the time constraint, a computing environment is forced to establish the trustworthiness of a program without going through the traditional evaluation cycle. In fact, the time spent on trust establishment should be only a small fraction of the brief execution time of the execution unit. Therefore, the third fundamental challenge of mobile code security is the following:

**Implicit Acquisition:** In the absence of an explicit acquisition process, how do we automate trust establishment in a limited time?

#### 3.4.4 A Subleasing Analogy

The situation can be clarified by considering the following analogy. Traditional multi-programming systems are in some ways like a residential building (a host). A landlord (operating system) decides to rent the property to multiple tenants (users/processes). The landlord defines access policies to prevent tenants from interfering with each other, and from unauthorized access to facilities in the building. Also, according to the identity of the tenants (probably some tenants pay more rent than the others), they are given different degree of privileges to access the facilities of the building (e.g. some have to share bathrooms with others, while more privileged ones have their private bathrooms). As such, the authorization of rights is based on the landlord's familiarity with the tenants.

Mobile code computing is like the rental of business properties. Now the tenants (web browser processes) rent office space from the landlord to conduct business. The tenants want to make the office publicly accessible, so that customers (mobile programs) interested in using the tenants' services can come over to the offices. This complicates the security model in three ways.

Firstly, the customers are not familiar parties. Neither the landlord nor the tenants can naively base access authorization on the identity of the customers. Such practice will unnecessarily reduce the accessibility of business. This corresponds to the challenge of establishing *anonymous trust*

Secondly, each individual business should be able to define their own security policy on top of the security policy already imposed on them by the landlord. On one hand, individual business should make sure that the security policy defined by the landlord is honored by the customer who are accessing the office. For example, if the landlord requires that the building be closed after 9pm, then the tenants should make sure that the customers leave before then. On the other hand, the business should have mechanisms to enforce additional security measures for their customers. For example, the office equipments should be operated only by the employees of the business, and should be guarded from unauthorized access by the customers. In summary, *layered protection* is needed in this situation.

Thirdly, customers don't wait. Security measures may not interfere with the brief visitation of the customers. Individual business are only granted limited time to enforce security. This corresponds to *implicit acquisition*.

What characterizes the above situation is the phenomenon of *subleasing*. The operating system leases resources to a trusted group of users. In turn, a special group of users, the mobile code computing environments, sublease their share of resources to mobile programs with unknown origin. Also, the visitation of mobile code is by nature brief and transient. Any competent mobile code security model should attempt to address the challenges created by this subleasing phenomenon.

In summary, code mobility defines a new software engineering model: the new reality of brandless programs necessitates anonymous trust; the need for protecting mutually distrusting code from each other requires layered protection within a single process; implicit acquisition forces the software evaluation process to be time-constrained and automatable.

### 3.5 Software-based Solutions

There is a misconception that hardware-based protection mechanisms are the only way to ensure system integrity [10]. Though some of the security requirements in mobile code systems can potentially be addressed by hardware solution, we here focus on software-based solutions which implement logical separation (section 3.2, page 11). We do this for the following reasons:

1. **Portability.** Execution units might migrate to a heterogeneous array of computing platforms. A hardware solution limits the kind of platform an execution unit may be executed on. We target protection mechanisms that can be implemented in general-purpose processors, on typical operating systems.
2. **Performance.** Numerous reports [121, 105, 122] confirm that memory protection schemes based on hardware-enforced address space boundary introduce significant performance penalty for cross-boundary procedure calls. Software-based memory protection mechanisms like Software-based Fault Isolation [121], Proof-Carrying Code [77], and type-safe languages [11] are found to be a lot cheaper than hardware-based mechanisms. Even interpreted languages like Java are also found to exhibit adequate performance for system programming [92]. All these suggest that software-based solutions can be as adequate as, and at times even more competitive than, hardware-based solutions.
3. **Ease of Experimentation.** A software solution is usually easier to implement and modify. It can then be easily distributed to or reproduced by the computing community, who in turn may iteratively review and improve the solution.
4. **Expressiveness.** Though memory protection naturally invites hardware solution, it is usually difficult to reduce high-level access control mechanisms into hardware implementation, especially when we are dealing with application-domain-specific security concerns.

In the following, we will focus mainly on software-based solutions to mobile code security.

## 4 Protection Mechanisms for Mobile Code Systems

In this section we outline four software-based approaches for protection: *discretion*, *verification*, *transformation*, and *arbitration*. The approaches presented here are complementary. In fact, most of the protection mechanisms of existing mobile code systems can be understood as a combination of mechanisms from our list.

### 4.1 Discretion

*Guard: Okay, stop right there! This wing has limited access. Show me your badge.*

*Intruder: Here it is.*

*Guard: Oh, Mr. President, please come in...*

Discretion refers to protection mechanisms that relies on “tokens” of trust to make security decision. In particular, we refer to the use of various authentication techniques [101, 60] for establishing trust. Every mobile code unit is associated with some digital signature(s). Whenever a foreign mobile code unit arrives at a host, its signature is authenticated, and a (mechanical) process of authorization will translate the result of authentication into access privileges. In such systems, signature authentication is assumed to be very efficient. Because of its inherent simplicity and the efficiency of signature verification, discretion-based protection addresses the challenge of implicit acquisition very well. As a result, it has been studied as a general protection infrastructure [38, 56] and is implemented in many existing mobile code systems (e.g. Java [58, 47, 46, 48, 7, 49], Telescript [125, 110], Agent Tcl [51], ActiveX [112, 114], etc).

At the heart of the discretion approach is the *semantics* of the signature. What a signature means determines the kind of access rights granted. In the following, we survey issues surrounding the assignment of meaning to a signature.

#### 4.1.1 Multiple Trust Levels

An authorization procedure maps signatures to a space of meaning. The granularity of control depends on the size of the meaning space. A large meaning space results in finer-grained control. We illustrate this by the following example.

The ActiveX approach [112, 114] relies solely on authentication to enforce security. ActiveX controls are native mobile code. An ActiveX control is transported with a signature that identifies its origin. It is up to the user to decide if the signature is trusted, and, in cases when the foreign control is trusted, it will be granted full access to the host.

Authentication technologies, both cryptographic and non-cryptographic [101, 60], are well established. In addition to its inherent simplicity, the ActiveX approach results in minimal interference and allows execution units to have maximal capability. This represents a property of nearly all discretion-based protection mechanisms: signature checking reduces the need for run-time or link-time interference.

The ActiveX approach grants trusted code with unlimited access to the entire host environment. This represents an “*all-or-nothing*” model of security. In many cases,

this inflexibility is absolutely unacceptable. For example, we might want to grant an execution unit rights to read local files as long as it is forbidden to open socket connection to remote sites (and thus disclose contents of our local files). Therefore, when we talk about security, there are actually different levels of trust. Based on the result of authentication, we might want to grant certain execution units more capabilities, while granting less to the less trusted execution units. A discretion-based protection mechanism must provide fine-grained authorization mechanisms.

The need for fine-grained access control is epitomized by the evolution of the sandbox model in Java [48]. In the 1.0.2 version of the Java Development Kit (JDK), all code are untrusted unless it is loaded from the local disk. Untrusted code is executed in a sandbox in which access to most system resources is prohibited. In JDK 1.1, signed applets are introduced. Applets associated with a trusted signature is given full access to the system, while the untrusted applets remain in the sandbox. In the coming JDK 1.2, extensive access domain architecture be incorporated to the security infrastructure, allowing the selective mapping of signatures to collections of permissions. Thus, one may define “sandboxes” of different access restrictions, and put applets of various trust levels into appropriate sandboxes.

#### 4.1.2 Denotation of Signatures

A digital signature is an unforgeable token that denotes a security property of the signed code unit. Three potential denotations can be attached to signatures of mobile code units.

**Identity/Origin Semantics:** The signature of a mobile code unit identifies its author or origin. A computing environment maintains a mapping between known signatures and their associated rights. This method is a direct translation of the traditional identity-based authorization scheme found in most operating systems. As we have discussed in section 3.4.1, schemes as such that are based on knowing the programs’ owner does not work well in the establishment of anonymous trust.

**Authoritative Endorsement Semantics:** Signing a mobile code unit means that the signing party endorses the unit as being “safe”, usually in an informal sense. In such an approach, some trusted authority will be responsible for certifying mobile code units. A programmer submits his/her mobile program to the certification authority before the program’s publication. Usually, what it means to be “safe” is defined informally, if it is properly defined at all. In such approaches, a signature signifies nothing more than the endorsement of the mobile code unit by a certain party. Such endorsement has no formal semantics — it cannot be reduced to formally defined security properties. The entire notion of endorsement is based on trust. Having Microsoft endorsing a piece of software does not necessarily imply its being bug-free. Notice, however, authoritative endorsement is more or less how our consumer economy works (i.e. brand names and such).

**Program-Analytic Semantics:** The signature means a formal program-analytic property like type-safety or program invariants. A signature is attached to the mobile

code unit only when the corresponding formal property is found in the unit. The attachment of signature can result from three possibilities:

1. Code is trusted because it is *generated* by a trusted *compiler* [94, 78].
2. Code is trusted because it has been properly *rewritten* by a trusted program *transformer* [102].
3. Code is trusted because it has been *certified* by a trusted program *analyzer* [35].

A program-analytic semantics may be more reliable than informal endorsement, because we are now trusting a formally defined, publicly available program certifying algorithm instead of mere human judgment. Right now, there are not many security properties that have been formalized into program-analytic terms. Memory safety and confidentiality are the rare cases that has been formalized (see section 4.2). Works on the translation of security properties into program-analytic terms will definite further the feasibility of this approach.

There are two inherent challenges to adopting a program-analytic semantics for digital signature:

- Suppose a flaw is found in a widely accessible implementation of a program certifying algorithm, or, say, such an implementation no longer reflects the evolved security policy. In such cases, we need to be able to revoke the key that generates the signature.
- We need to guarantee that signatures are not forgeable, that is, only the program certifier we trust may sign mobile code units. This problem is more complicated than it seems, especially when we are to distribute such trusted program certifiers the hands of untrusted programmers. How can we be sure that the certifier is not tampered with?

In section 4.2.4 we will discuss how trusted hardware and key-management schemes are used to address the above challenges.

### 4.1.3 Evaluation

- Authentication is efficient and fully automatable, and thus addresses the challenge of implicit acquisition very nicely.
- Depending on the denotation of the signature, the discretion approach might or might not address the challenge of anonymous trust well.

## 4.2 Verification

*Guard: Okay, stop right there! Please go through the metal detector.*

*Intruder: ...*

*[Alarm goes off ...]*

A *firewall* [88] is a first degree approximation to the *static verification* approach. A firewall is a computer that sits between a local network and the rest of the global network. It filters packets as they go by, according to various criteria which can be configured. Packets that appear unsafe are rejected before they can do any harm to the protected network.

In the verification approach to mobile code security, security policies are formulated as program analytic properties. Incoming mobile code units must pass through a trusted program analyzer before reaching the computation environment. The trusted program analyzer, usually called a *verifier*, filters out potentially unsafe programs. The execution units that can reach the computation environment are guaranteed to satisfy certain security properties. Usually, there is no further need for the computation environment to provide any countermeasure to enforce the checked properties. Because of this, static verification is an attractive protection mechanism when the program property of interest is expensive to enforce dynamically (e.g. dynamic type checking).

#### 4.2.1 Verification for Memory Protection

To date, the most successful application of the verification approach is for memory protection. Here we use three examples for illustration.

**Typed Intermediate Language.** In the Java language [50], memory protection is achieved by the use of a safe intermediate language. Java source programs are compiled into *Java Virtual Machine (JVM)* bytecode [65]. The bytecode representation is specially designed to protect execution units from interfering with each other and from accessing the JVM’s internal state. Firstly, JVM bytecode is strictly typed. Secondly, it does not allow pointer arithmetic. Therefore, Java bytecode can only access memory it knows of in a type-safe manner. As a consequence, memory protection is reduced to type-checking. All untrusted classfiles (Java’s version of a mobile code unit) must pass through a bytecode type-checker (i.e. the Java bytecode verifier) before dynamically linked into the JVM. Since the JVM bytecode is unstructured, data-flow analysis must be used to make sure that the classfile is type-safe. In fact, the data-flow analysis does not only guarantee type-safety, it also checks for properties like stack<sup>5</sup> overflow. Therefore, at run-time, certain kinds of overhead, like run-time type-checking and stack overflow checking, can be safely avoided.

**Proof-Carrying Machine Code.** Type systems can capture only certain kinds of security properties, and occasionally requires a safe (intermediate) language. To enforce memory safety in a less structured language (e.g. native code), and to provide more expressive means for formulating safety properties, Hoare logic [27] can be used. Hoare logic is an axiomatic system for proving program correctness. Safety properties can be encoded as precondition/postcondition pairs, while verification reduces to the establishment of the postcondition given the validity of the preconditions.

In their work on proof-carrying code<sup>6</sup>, Necula and Lee uses Hoare logic to establish the memory safety of operating system kernel extensions [77] and native code ML library routines [76]. In the following, we use the example from their kernel extension work [77] to illustrate their approach.

---

<sup>5</sup>The stack we mention here is a evaluation stack local to a call frame, and is used for expression evaluation.

<sup>6</sup>To us the contribution of Necula and Lee’s work is twofolded. One is the demonstration of the feasibility of using Hoare logic as a practical verification formalism for mobile code security, and the other is the proposal of proof-carrying code as a verification protocol. For now, we are not concerned with the latter idea. We will return to that when we explore the design space of verification protocol (section 4.2.4)

Suppose a kernel maintains a table of descriptors. Each entry of the table is composed of two words: the first one is a tag specifying if the second data word is modifiable. The kernel allows users to install their application-specific accessing functions for the table entries. These accessing functions are passed pointers to a table entry when called. The functions must obey the following constraints:

- [C1 ] The user must not access table entries other than the one passed as argument.
- [C2 ] The tag word is read only.
- [C3 ] The data word is writable if and only if the tag is non-zero.
- [C4 ] Privileged registers must not be modified (remember that the access functions are called by the kernel and thus are run in kernel mode).

The verification of the above policy is performed in the following steps:

1. The operational semantics of the underlying instruction set is formalized into proof rules.
2. The policy is encoded as precondition and postcondition.
3. The code for the accessing function is translated into an array of predicates describing the effect of executing the code.
4. The conjunction of the precondition, postcondition, and the predicates in the last step forms a *verification condition*.
5. The proof that the verification condition follows from the proof rules in step 1 is attached to the code unit.
6. Any party who would be interested in executing the code unit must check if the attached proof correctly establishes the verification condition.

Proving the memory protection policies described above turns out to be a very tractable task, especially for certain low level routines (e.g. network packet filers) in which no looping is involved. Though some of the above constraints can be readily enforced by type systems in an abstract intermediate language (e.g. Java bytecode), Hoare logic offers flexibility that allows one to deal directly with native code instead of using compiled high level language or interpreted bytecode, both introducing considerable run-time interference that cannot be tolerated in certain applications<sup>7</sup>. Moreover, its ability to deal with constraint C4, in which the modifiability of one field is dependent on the value of another, demonstrates the expressiveness of Hoare logic. This benefit of expressiveness was demonstrated more fully in a later paper [79], in which they show how Hoare logic can be used to enforce some form of liveness condition, resource usage limitation, and data abstraction in an information harvesting agent.

The down side of using a proof system as powerful as Hoare logic is its inherent undecidability. In Necula and Lee's experience, the added flexibility may result in difficulties when dealing with looping programs [77]. In those cases, they might have to generate a proof by hand. To deal with this, they have recently experimented with automatically generating loop invariants by a certifying compiler [78]. In particular, they start with a small type-safe source language, and build a compiler that generates

---

<sup>7</sup>It is observable that PCC has significant performance advantages over its competitors like software-based fault isolation (section 4.3).

machine code annotated with typing specification and loop invariants in the mist of aggressive compiler optimization (e.g. array-bound checking elimination). For simple memory- and type- safety properties, the above approach effectively automates the generation of loop invariants. Yet, for the general case, manual intervention cannot be avoided without the use of some domain-specific language [79, Section 9]. This need for human assisted proof generation necessitates the use of proof-carrying code, which we will describe in section 4.2.4.

**Typed Assembly Language** While Java has to resort to an intermediate language in order to carry type information, and while Nacula and Lee have to resort to a highly expressive logical proof to capture similar kind of information for machine code, Morrisett *et al* [74, 75, 73, 44] recently demonstrate that type checking can actually be performed on an assembly language. In particular, they proposed a typed assembly language (TAL) [75] which carries the type information of a rich, functional source language (a call-by-value variant of System  $F$ , the polymorphic  $\lambda$ -calculus augmented with products and recursion on terms). The significance of their work is threefold. Firstly, their work demonstrates that type safety can be achieved without using an abstract intermediate language, which inevitably reduces run-time performance. The typed assembly code can be type-checked all by itself without the availability of the source program. Secondly, the typing construct imposes almost no restrictions on optimization. This makes the safety of the program independent of the compiler that generates the code. Thirdly, they present a type-preserving, effective procedure that translate the source language to TAL. This contrast with the work of Nacula and Lee [77], in which incompleteness is resulted due to the use of a more powerful verification system.

The discussion in this section brings out the following design principle:

**Typing as Memory Protection:** Memory protection can be captured as type systems[100, 14]; static type-checking can then be used to screen out unsafe mobile code units, thus avoiding run-time checks.

One may see Java bytecode as a portable intermediate representation which allows type annotation to be attached for the sake of enforcing memory protection statically. Proof-carrying code, when applied solely to memory protection, captures typing information for a target language using a very expressive logic, thereby providing static typing without using an interpretive intermediate language. Finally, TAL demonstrates that static typing can actually be performed in a target language without the use of an overly expressive formalism.

#### 4.2.2 Verification for Confidentiality

Program-analytic approaches to the enforcement of confidentiality have received a lot of attention, and are relatively well-understood. Building on Bell and La Padula's security model [9], the work of Dorothy Denning [31, 33, 32] has laid the foundation for the study of secure information flow analysis. Developments have been constantly reported [6, 71, 70, 72, 8, 1, 53]. In particular, the work of Volpano *et al* [120, 118, 117, 106, 119] on using a type system to capture information flow has recently attracted considerable attention from the mobile code community. In this section, we will give a brief survey



of Denning's lattice model of information flow, and the basic idea of Volpano *et al*'s type system.

In the US military's *multilevel security* model, documents are *classified* into a finite set of sensitivity levels such as *unclassified*, *restricted*, *confidential*, *secret*, and *top secret*. All personnel is granted a single *clearance* level. Clearance indicate the level of trust that is granted. Personnel with clearance *secret* may access all documents except those classified as *top secret*. The Bell-LaPadula model and its later generalization by Denning is a formalization of the above concepts. A security system is composed of a set  $S$  of subjects and a disjoint set  $O$  of objects. Each subject  $s \in S$  is associated with a fixed security class  $C(s)$ , denoting its clearance. Likewise, each object  $o \in O$  is associated with a fixed security class  $C(o)$ , denoting its classification level. The security classes are *partially ordered* by a relation  $\leq$ , and  $\leq$  forms a *lattice*<sup>8</sup>. To avoid subjects with low clearance accessing sensitive data, we need the following property:

**Simple Security Property.** A subject  $s$  may have *read* access to an object  $o$  only if  $C(o) \leq C(s)$ .

To avoid subjects with high clearance to release sensitive data to low-clearance subjects, we need the following property:

**\*-Property.** A subject  $s$  who has read access to an object  $o$  may have write access to an object  $p$  only if  $C(o) \leq C(p)$ .

In summary, a subject may only read objects with classification level no higher than its clearance, but may only write to objects with classification level no lower than its clearance. Information is always unidirectionally flowing from low classification source to high classification destination.

In the context of mobile programs, one may want to control the flow of information among variables and routines. For example, one may want to protect the content of a private variable from leaking into globally accessible data area, or from flowing into a routine that writes to a socket. To illustrate this, let us define a very simple procedural language:

$$\begin{aligned} (\text{expressions}) \quad e &::= x \mid n \mid e + e' \mid e = e' \mid e < e' \\ (\text{commands}) \quad c &::= e := e' \mid c; c' \mid \text{if } e \text{ then } c \text{ else } c' \mid \\ &\quad \text{while } e \text{ do } c \mid \text{letvar } x := e \text{ in } c \end{aligned}$$

Suppose we have two variables  $H$  and  $L$ , so that  $H$  contains some classified data and  $L$  is globally accessible. Information about the value stored in  $H$  can flow into  $L$  in at least two ways. Firstly, information flow may be *explicit*:

$$X := H + 0; L := X;$$

Secondly, information flow from  $H$  to  $L$  may also be *implicit*. Conditional statements may convert information into control flow.

$$\begin{aligned} &\text{if } H > 0 \text{ then} \\ &\quad L := 1; \\ &\text{else} \end{aligned}$$


---

<sup>8</sup>A partial ordering is a relation which is transitive and antisymmetric. A lattice is a partial ordering in which every pair of elements possess an upper bound and a lower bound. Consult Davey and Priestley [29] for more information.

```

    L := 0;
  end if

```

Likewise, looping constructs can implement counting effectively.

```

    L := 0;
  while H > 0 do
    H := H - 1;
    L := L + 1;
  end while

```

Information flow analysis attempts to statically uncover these kinds of information leakage. As noted by Podgurski and Clarke [89], information flow analysis is a kind of dependence analysis [41]. In the following, we use a type system to capture program dependence.

To deal with explicit information flow, each expression is associated with a secure flow type, which represents the classification level of the data item. The lattice structure of the classification levels induces a natural subtyping relationship among the secure flow types: if type  $\tau$  represents a classification level as high as that of type  $\tau'$  then  $\tau \geq \tau'$ . An expression involving operands with distinct security types receives the least upper bound of the operands' types as its type. For example, if  $e$  and  $e'$  have security types  $\tau$  and  $\tau'$  respectively, and that  $\tau \leq \tau'$ , then  $e + e'$  can be assigned security type  $\tau'$ . Each variable also has a type  $\tau$  *var*, indicating that it holds contents with type no higher than  $\tau$ . Explicit leaking is then prevented by requiring that assignment of the form  $X := a$  is well-typed only if  $X$  has type  $\tau$  *var* and  $a$  has type no higher than  $\tau$ . To formally express this, we allow expression type  $\tau$  to be coerced to any type  $\tau'$  if  $\tau \leq \tau'$ , and then require that  $X := a$  is well-typed if and only if  $X$  has type  $\tau$  *var* and  $a$  has type  $\tau$ . With this arrangement, the above example code that explicitly leaks information will not be well-typed.

To handle implicit information flow, every command is associated with a type  $\tau$  *com*. Intuitively, a command has type  $\tau$  *com* if every variable that is being assigned in the command has type  $\tau'$  *var* where  $\tau \leq \tau'$ . That is,  $\tau$  is a lower bound for the security levels of the variables being assigned in the command. The idea is that if a conditional or iterative construct involves a condition expression of type  $\tau$  then commands in the body should not assign to variables with security levels lower than  $\tau$ . To make this work, we need two more subtyping rules. For the variables,  $\tau$  *var*  $\leq$   $\tau'$  *var* if and only if  $\tau \leq \tau'$ . For the commands, the opposite must hold:  $\tau$  *com*  $\leq$   $\tau'$  *com* if and only if  $\tau' \leq \tau$ . Again, expressions can be freely coerced to their supertypes. The following type rules are then defined:

- An assignment statement of the form  $X := a$  has type  $\tau$  *com* if  $X$  has type  $\tau$  *var* and  $a$  has type  $\tau$ .
- A sequential composition of the form  $c; c'$  has type  $\tau$  *com* if both  $c$  and  $c'$  have type  $\tau$  *com*.
- A conditional statement of the form **if**  $e$  **then**  $c$  **else**  $c'$  has type  $\tau$  *com* if  $e$  has type  $\tau$  while  $c$  and  $c'$  have type  $\tau$  *com*.
- An iterative statement of the form **while**  $e$  **do**  $c$  has type  $\tau$  *com* if  $e$  has type  $\tau$  while  $c$  has type  $\tau$  *com*.

Volpano *et al* prove that a type system like the above satisfies variants of the simple security property and the \*-property [120]. They also extend their type system to account for covert flow [117], procedures [118], and concurrency [106]. A *covert channel* is a mechanism which is not intended for communication but, nevertheless, may leak information. A good example is that a thread  $T$  may be constructed to conditionally enter an infinite loop depending on the value of a classified variable. Now another spying thread  $S$  may then time the execution of thread  $T$ , thereby obtaining information about the classified variable without exchanging data with  $T$ . To deal with procedures, they use a constrained type to capture the condition on which the procedure may be executed securely. Instead of a fixed type, a principal type containing subtype inequalities is computed. They have also studied the impact of various scheduling assumptions on the soundness of secure flow types in the presence of concurrency. It is shown that the original type system in [106] has to be properly restricted in order for soundness to be preserved (e.g. by restricting the kind of condition that can appear in a conditional or iterative statement).

Information flow analysis is a very difficult problem. In a simple sequential language in which there is no malicious third-party observer, information flow analysis as described in this section is fully adequate. However, in a context typical to mobile programs, neither the sequential assumption nor the closed-world assumption hold. In order to deal with the complication of covert channels and malicious observers, analysis like the above quickly becomes overly conservative to be useful. Recent development in this area attempts to provide a more accurate analysis. Heintze and Riecke [53] propose a typed  $\lambda$ -calculus, SLam calculus, with a type system that combines both secure information flow analysis with trust analysis (see below). They achieved a finer-grained analysis than previous approaches by representing implicit and explicit information flow separately.

### 4.2.3 Verification for Integrity

Although confidentiality has been a relatively well-understood security need, less attention has been given to data and system integrity. In fact, there has been no consensus definition for integrity [99, 67].

The earliest proposed integrity model is that of Biba [12]. It is more or less a direct translations of Bell-LaPadula's lattice model to the domain of integrity. In particular, one wants to avoid untrusted data from flowing into trusted output. To enforce this, the output of an operation involving both trusted and untrusted parameters must be regarded as *untrusted*. Level of trust is modeled by a lattice of integrity labels. Information can only flow unidirectionally from high integrity source to low integrity destination. The requirement is formalized as a *simple integrity property* and an *integrity \*-property*, analogous to their counterparts in the Bell-LaPadua model, of which Biba's integrity model is simply a dual.

Implementing the above idea, Ørbaek [81] defines an instrumented semantics to capture information flow in a small imperative language. The language offers constructors for trusted data and untrusted data. In the instrumented semantics, values are tagged with a flag indicating their trust levels. The trust levels of values reduce as they are combined with values of lower trust levels. Ørbaek defines a tractable, abstract interpretation of the language that guarantees safety. He also proposes a con-

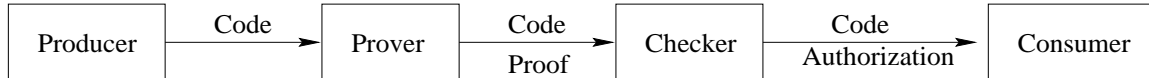


Figure 3: Participants of a Verification Protocol

straint generation mechanism that builds a partial ordering of the variables appearing in a program. A solution to the constraint system assigns appropriate trust levels to the variables. The constraint-based analysis allows program modules to be analyzed separately. He later extends trust analysis to a higher-order  $\lambda$ -calculus [85, 82].

Since the inversion of a lattice is still a lattice, a protection mechanism that can enforce confidentiality in Bell-LaPadula’s sense can also enforce integrity in Biba’s sense. Since the product of two lattices is also a lattice, no new mechanism is needed even if one is to enforce both the Bell-LaPadula and the Biba models. Based on the above observations, Sandhu [99] has rightly concluded that only a restricted aspect of integrity is captured by directional control of information flow, and that this restricted aspect has already been captured in the Bell-LaPadula model. Many have observed that the security requirements of commercial applications and some emerging military applications, which are more concerned with integrity than confidentiality, is very different from the security requirements of traditional military applications, in which confidentiality dominates [25, 67]. Compared to confidentiality, integrity is a less well-defined notion. It is difficult to formally differentiate a legitimate data modification from a malicious attack. Teleological knowledge about the application domain has to come into the picture. Dealing with this gap, the Well-formed Transaction model of integrity has been proposed by Clark and Wilson in 1987 [25]. It is an interesting research direction to see if Clark and Wilson’s model can be translated to program-analytic terms, and then applied to integrity enforcement in mobile code systems.

#### 4.2.4 Verification Protocol

The verification approach requires only that security properties of the mobile code units are checked prior to their execution. It does not specify how the various parties involved in the verification process are to be orchestrated. In the following, we will see that the design space within the verification approach is in fact much wider than we bet on earlier. We use the term *verification protocols* to refer to schemes that specify how various concerned parties cooperate to carry out a verification procedure. A typical verification system consists of four parties (see figure 3):

**Producer:** The party who generates the mobile code unit.

**Prover:** The party who takes a mobile code unit, and generates a proof that the code satisfies a predefined set of security properties.

**Checker:** The party who takes a mobile code unit and the proof generated by the prover, and translates it into a decision of whether execution/linking is authorized.

**Consumer:** The party who takes the mobile code unit and an authorization of execution, and executes the code.

A verification protocol specifies where the above participants are located, and how they orchestrate the verification process. We shall overview four verification protocols:

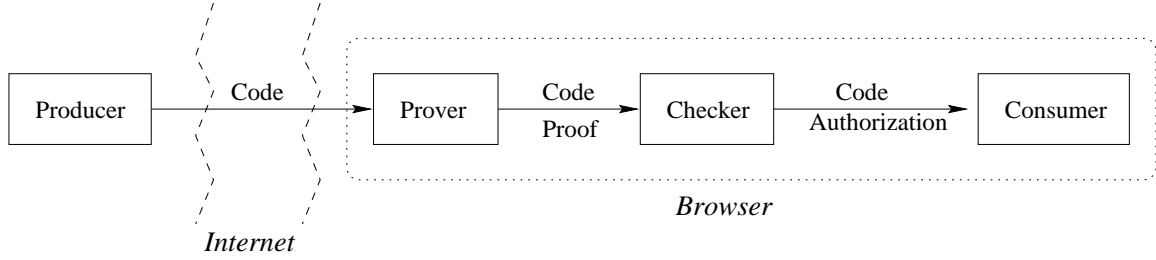


Figure 4: The Proof-on-Demand Protocol

*proof-on-demand, proof-carrying code, proof delegation, and proof linking.*

**Proof-on-Demand.** The proof-on-demand protocol is exemplified in the Java mobile code system. Both the prover and the checker are on the host that execute the mobile code unit (see figure 4). Whenever an untrusted classfile is to be linked into the JVM, the bytecode verifier will be invoked on-the-fly to prove that the classfile is structurally intact and type-safe [65]. Since the JVM assumes both the prover and consumer role, there is no need to generate the proof explicitly, and thus proof checking reduces to the simple authorization of execution if verification succeeds.

Performing verification in the computing environment has the key advantage that it allows run-time code generation. Execution units may dynamically generate code that will be linked in on-the-fly. Having the verifier built inside the dynamic linker allows such dynamically generated code to be verified properly.

However, the proof-on-demand protocol has several drawbacks:

1. Compared to authentication-based approaches, proof-on-demand introduces extra link-time overhead to the computing environment. For example, in the case of Java, verification could involve complicated data-flow analysis, which introduces observable slow down to the dynamic linking process<sup>9</sup>.
2. There is a close coupling between the verifier and the computing environment. Such close coupling introduces two software engineering problems:
  - (a) When a bug is found in the verifier, as frequently happened in various industrial strength Java-enabled web browsers<sup>10</sup>, one has to replace the entire computing environment (e.g. browser). This is usually difficult to guarantee because there are in an order of  $10^7$  browser users out there, and it is difficult to guarantee that they are all updated<sup>11</sup>.
  - (b) If the architecture of the computing environment is not well articulated, there will be many dependencies among the loader, linker, and verifier. Consequently, building a correct verifier is made more difficult.

<sup>9</sup>McGraw and Felten [68, p 110] observe that the slow down could be as unbearable as a denial-of-service attack.

<sup>10</sup>Consult [30, 68, 57, 115, 113] for a shamefully endless list of bugs found in various web browsers.

<sup>11</sup>When is the last time you upgrade your browser? Is your browser the most up-to-date version?

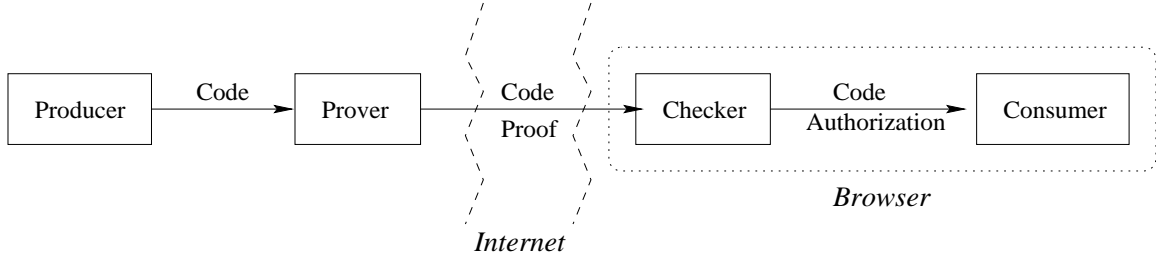


Figure 5: The Proof-Carrying Code Protocol

**Proof-Carrying Code.** In the proof-carrying code (PCC) protocol [77, 76], the verifier is located at the producer side of the internet (see figure 5). Verification goes in the following steps:

1. The consumer publicizes a safety policy in the form of some program-analytic properties. The properties should be specified in a way so that a proof that the properties are present in a program can be formally expressed and mechanically verified.
2. If a producer of code wants to ship code to the consumer, it must first acquire the safety policy.
3. On behalf of the producer, a prover then produce, manually or mechanically, a proof that the code to be shipped satisfies the consumer’s safety policy.
4. The proof is then attached to the code when it is shipped.
5. When the code and the proof arrive at the consumer side, a mechanical proof checker will determine if the proof establishes the safety of the code. If so, it authorizes execution.
6. The consumer executes the code.

The idea of proof-carrying code is based on the intuition that proof-checking is much easier than proof-generation. Firstly, since the safety of the mobile code unit is proven on the producer side instead of being repeated every time the code is loaded into the computing environment (as in the case of Java), the protocol effectively reduces link-time interference. Secondly, transferring the “burden of proof” to the code producer allows one to consider more expressive safety policy. The consumer does not have to worry that verifying such policy will introduce unbearable performance overhead to linking, for only proof checking is involved. With this setup, even manual proof generation is affordable assuming standardized policies for consumers. The resource the producer put into generating a safety proof for the code will be amortized over multiple use of the code.

A potential problem of proof-carrying code has to be mentioned. In the worst case, the size of the proof may be exponential to the size of the program [77, 79]. This creates a problem for the transportation and checking of the generated proof. However, Necula and Lee observe that such “proof bloat” is a rare case.

Recent development of the idea of PCC includes Rose and Rose’s work on lightweight bytecode verification [93] and Morrisett *et al*’s work on typed assembly language [75]. The common theme is to avoid the full expressiveness of proof-carrying code, while

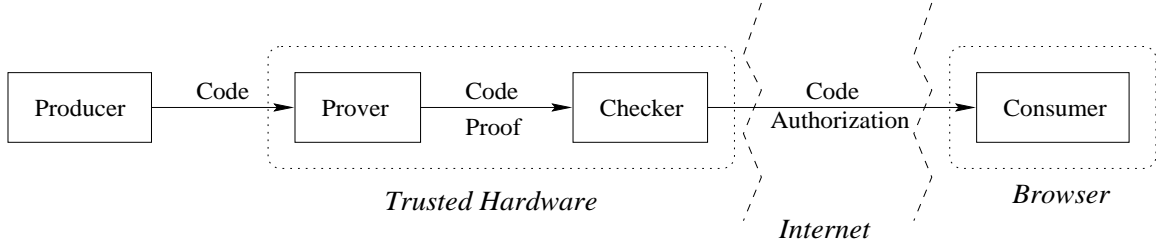


Figure 6: The Proof Delegation Protocol

concentrating on a specific program analysis problem (e.g. type checking), in which small certificates can be generated mechanically by a certifying compiler. In general, a research direction is to understand how results of various specific static analyses can be captured as *certificates*, so that:

1. The consumer of a piece of untrusted code can bypass the static analysis partially or entirely. The certificate witnesses to program properties the original analysis attempts to uncover.
2. The time spent to check the integrity of a certificate is significantly smaller than that of performing the analysis.
3. The certificate should be small in comparison with the size of the program it describes.

**Proof Delegation.** Proof delegation<sup>12</sup> [36, 35] is motivated by a number of security and software engineering concerns, including the following:

1. **Performance:** Verification, be it proof generation or proof checking, introduce performance degradation. Can we remove *either* or even *both* from the consumer’s burden?
2. **Disclosure:** Most of the verification protocol involves the use of a high level (intermediate) language (proof-on-demand) or the attachment of program annotation (proof-carrying code). Such representation discloses intellectual properties that the producer might consider proprietary<sup>13</sup>.
3. **Configuration Management:** In all verification protocols, a verifier of some sort is involved. The implementation of such program verifier is usually complicated, and bugs are found in commercial implementations (see footnote 10). When verifier bugs are found, how easy is it to distribute the upgrades? Alternatively, how can customers be protected from using “outdated” computing environment with known security vulnerabilities?

The proof delegation protocol consists of three components (see figure 6):

1. **Trusted Hardware:** Beside publicizing the theorem to be proven, the consumer also pass to the producer a trusted proof generator and a proof checker. The target theorem, the proof generator, and the proof checker are all installed into some

<sup>12</sup>The term “proof delegation” was not coined in the original papers. We coin the term simply because it reflects accurately where the approach lies within the wide spectrum of verification protocols.

<sup>13</sup>It is well known that Java bytecode can be decompiled readily [69].

trusted coprocessor, and the entire package is handed to the producer. Trusted coprocessors are usually realized in the form of hardware extensions to consumer equipment, for example, as PCI cards [55] or PCMCIA cards [107, 24]. They contain a secret key, which is physically protected from tampering<sup>14</sup>. When a code producer wishes to distribute a mobile program, he submit the code to the trusted coprocessor. The embedded verification software in the trusted hardware checks if the code is safe, and then attaches a digital signature to the code using the secret key in the trusted coprocessor. The producer can then distribute the properly signed code.

2. **Authentication-enabled Computing Environment:** A computing environment that has authentication capability will then be able to check if a mobile code unit has been endorsed by a trusted coprocessor. If so, no link-time check will be necessary.
3. **Key Management Infrastructure:** When faults are found in an implementation of the verification algorithms, the key associated to the implementation can be revoked automatically by well-known key revocation techniques. From then on, mobile code units signed by the expired key (corresponding to the faulty implementation) will have to be re-certified. Browser owners have no need to upgrade their browsers.

In effect, the verification effort is delegated to a trusted program analyzer. Firstly, on the customer’s side, proof generation and proof checking is replaced by efficient signature checking, thereby reducing *performance* penalty. Secondly, no program analytic annotation is needed for the authorization of execution, thereby avoiding *disclosure* of intellectual property. Thirdly, when a verifier implementation is found to be faulty and its corresponding key revoked, there is no need to force all the users of computing environment (e.g. browsers) to upgrade their software. Only the mobile code producers are affect. This is much more manageable, for the number of producers is significantly smaller than the number of consumers. Moreover, mobile program developers have an obvious motivation for actively updating their certifying hardware. This arrangement elegantly addresses the *configuration management* need of keeping verifiers up-to-date.

The use of trusted hardware establishes a physically secure binding between a signature and a well-defined, program analytic semantics (section 4.1.2). Such binding makes the distribution of trusted program certifier possible. Without such binding, a malicious programmer may tamper with a trusted certifier, generating signed mobile code units that are in fact faulty. But now the trusted certifier is protected by the trusted hardware, and such tampering will not be possible.

**Proof Linking.** Fong and Cameron [42] propose a verification protocol, proof linking, that enables separate verification in the presence of lazy, dynamic linking. The goal is to have modules in a mobile program verified separately, probably using different verification protocols, and then the results of verification combined safely at the execution site. To verify a module, information about other module may be needed due to inter-modular dependencies. However, such information may not be available

---

<sup>14</sup>Attempting to physically tamper with the hardware and the firmware will trigger protection mechanisms that erase the secret key on board.



due to various reasons. Designed to address this, the proof linking protocol organizes verification into two stages:

1. **Modular verification:** Each module is verified separately. Since the prover is not allowed to base its analysis of one module on the information found in other module, it performs partial verification. In particular, it defers the validation of dependency information by proving *commitments* that are predicated by *proof obligations*:
  - (a) At any point the prover requires information from another module, it formulates a *proof obligation* that captures properties assumed to be true in the other module.
  - (b) With the assumption of proof obligations, the prover will be able to establish certain properties. These properties are then asserted as *commitments*, which can later be used to discharge proof obligations of other modules.

At link-time, when a module is loaded and verified using some verification protocol, the obligations and commitments are exported as the result of verification. The resulting commitments are then asserted into a global deductive database, while the obligations are registered to be the guards of the module symbols they are protecting.

2. **Proof linking:** When a module symbol is to be resolved, its guarding obligations are looked up. The obligations are submitted to the commitment database as queries to be satisfied. If the obligations can be satisfied, then the symbol is successfully resolved; otherwise, proof linking fails, and the request of resolution is denied.

As a result of this set-up, when loading, verification, and linking happens asynchronously in the computing environment according to some predefined partial ordering, we have an *incremental* verification procedure. In order for this verification procedure to function properly, the proof linking protocol must satisfy several correctness conditions, namely, *safety*, *monotonicity*, and *completeness*. In their paper, Fong and Cameron formally defines these properties, and use them to establish the correctness of a proof linking version of Java bytecode verification.

Separate verification turns out to be vital to the adoption of such verification protocols as proof delegation [35], in which the verifier embedded in the trusted coprocessor can only access Java classfiles one at a time. More than that, separate verification offers interoperability among the three verification protocols we have mentioned. As we have seen, each verification protocol we presented above is useful on its own right: code-on-demand for dynamic code generation, proof-carrying code for extremely-hard-to-produce proofs, and proof-delegation for efficient, distributed, mechanical verification. With proof linking, one is then able to support all three in a computing environment. Since proof linking only depends on the correctness of proof obligations and commitments, the process that generates them can be any verification protocol.

Separate verification also results in a more maintainable computing environment. Existing Java verifier implementations lacks modularity. Because the verification of one classfile might require information of another classfile, the verifier might recursively invoke the loader to bring in the latter classfile. And to make sure that the newly downloaded classfile is providing correct information, some of the verification code

is located in the loader<sup>15</sup>. It is well-known in the program understanding literature that interleaving and delocalized program plans lead to programs that are difficult to comprehend [95, 64]. The proof linking architecture, when imposed on the Java verifier, gives modularity to its implementation by eliminating interleaving and delocalization, thus making it easier to maintain and to validate.

“passes”. Type-checking code is therefore spread all over the run-time system. It is well known in program understanding literature that interleaving and delocalized plans lead to programs that are difficult to comprehend [95, 64]. The proof linking architecture, when imposed on Java verification, reduces interleaving and delocalization in the Java verifier, making it easier to be maintained and validated.

#### 4.2.5 Evaluation

- Since safety properties to be statically verified are program analytic, they do not depend upon knowledge of the execution units’ origin, thereby addressing the challenge of anonymous trust very well.
- Since the computation environment has no need to re-establish the checked properties, run-time performance could increase drastically when equivalent dynamic checks are expensive. However, the checking of static properties could itself involve a very high cost. Depending on the exact verification protocol being used, some of the cost might be incurred at link-time.
- Static analysis is inherently limited. There are properties that cannot be verified statically, and there are others that do not allow efficient verification.

### 4.3 Transformation

*Guard: Okay, stop right there! Please go through the metal detector.*

*Intruder: ...*

*[Alarm goes off...]*

*Guard: Leave your Swiss Army knife here. I’ll give it back to you when you leave this building.*

A mobile code representation that is good for transportation (e.g. portable to many platforms) may not be tailored for execution. In many mobile code systems, mobile code units are transported in the form of virtual machine bytecode, and then, upon arrival to a host, the bytecode is transformed into native code for efficient execution. Such just-in-time (JIT) compilation is now a core feature of mobile code systems like Java [127], Omniware [66], and Juice [43]. Yet, dynamic code generation can also be seen as a means to protection. Mobile code units are expressed in a high level representation (e.g. a type safe intermediate language as in Java) in which unsafe behavior cannot be expressed. Upon arrival at the host, the code units are translated into a format which is directly executable on the host machine. Since code generation is performed by a trusted compiler on the host, and since unsafe behavior cannot be expressed in the source code, the generated code can be considered safe.

---

<sup>15</sup>This corresponds to the first pass of bytecode verification [65].

In similar vein, transformation can be used to tailor untrusted code into a more secure form. In contrast to dynamic code generation, unsafe behavior *can* be expressed in the migrated code. Yet, upon arrival at the host, the code unit is statically analyzed, and extra protection code is injected at program points where security cannot be guaranteed.

#### 4.3.1 Transformation for Memory Protection

In as early as the 1970's, code rewriting has been applied to memory protection within a single address space [34]. Recently, the Omniware mobile code system [66] uses transformation to implement memory protection for untrusted mobile code units. Omniware mobile code units are transported as bytecode for the Omniware Virtual Machine (OmniVM) [5]. OmniVM is designed to resemble a RISC architecture, thus allowing efficient performance, simple implementation, and retargetability. OmniVM divides its address space into segments. In order to ensure that execution units access only those segments they are authorized to, Software-based Fault Isolation (SFI) is used [121]. The basic idea of SFI is to rewrite untrusted mobile code units into versions containing no access to unauthorized segments. Each memory address is divided into two parts, namely, a segment identifier and an offset within the segment. Two possible rewriting rules can be formulated:

1. **Segment Matching:** For every memory reference, guarding code is inserted before the instruction that initiate the reference. The inserted code dynamically checks if the segment referred to is the same as the one the reference is located in. If it is not, then a memory fault is raised.
2. **Sandboxing:** For every memory reference, the segment identifier of the target address is dynamically replaced by the identifier of the segment the reference is located in.

The systematic application of either rule to every memory reference in a program guarantees that no interference would occur among disjoint segments.

Experience indicates that observable run-time overhead is caused by this approach because additional code is introduced by the transformation. Despite this overhead, native code that is instrumented this way can run at speeds comparable to the original code [121], although not as efficiently as a proof-carrying code version [77, 79].

In extensible operating systems VINO [102] and Exokernel [37], users are allowed to dynamically download untrusted extension code into the kernel's address space to modify the behavior of the operating systems. To avoid corruption of the kernel address space, untrusted extension code units are subject to SFI transformation before downloading.

#### 4.3.2 Transformation for Access Control

So far, the mobile code community has only explored the use of the transformation approach to the enforcement of memory protection. An interesting research question is how transformation might be used to establish higher-level access control.

Although their work does not elicit the full potential of the transformation approach, that is, rewriting unsafe programs to safe ones, Wallach and Felten [123] propose a transformation that addresses security issues in the context of access control in Java.

Access control can be performed in Java by run-time stack inspection (section 4.4.3). The Java API provides routines for traversing and examining the run-time stack [122]. Privileged service routines can utilize stack inspection to identify its callers in the call chain, and thus decide if access is granted. A caller may explicitly grant rights to a callee by appropriately annotating the current stack frame.

Wallach and Felten propose a transformation that rewrites Java code into what they call the *security-passing style*. All stack inspection operations are eliminated. In replacement, methods are provided with an extra parameter that encapsulates all the access control information inherited from the caller. All stack annotation operations are rewritten as operations on this extra parameter. When the transformation is systematically applied to Java bytecode before they are linked into the JVM, one will be able to execute Java programs written in stack-inspection style in a pure JVM without stack-inspection facilities. This is appealing because we can now express stack-inspection-based access control within the existing semantics of Java, and can thus avoid increasing the complexity of the JVM.

### 4.3.3 Evaluation

- The transformation approach shares many similar advantages with the verifier approach, including coping with anonymity. In addition, it is plausible that many of the verification protocols can be applied in the transformation approach. For example, rewriting may occur at a remote site, with the rewritten code bearing the signature of a trusted transformer, thereby utilizing the proof delegation protocol.
- Theoretically, transformation can be used to enforce other dynamic properties. Static analyses are usually very conservative. Many of the safety properties that one would like to check for are dynamic in nature. It is conceivable that many mobile code units rejected by certain static analyses can be “*made*” safe by injection of protection code.
- The transformation approach may introduce unnecessary run-time overhead.
- Transformation hinders code sharing within a computing environment. For example, two different transformations may need to be applied to the same piece of code in order to satisfy the distinct security needs of two different execution units. Consequently, two rewrites may have to be generated, one for each execution unit. These can be avoided only when the rewrite is uniform for all execution units, as in the cases of software-based fault isolation and security-passing style.

## 4.4 Arbitration

*Guard: Sir, this wing has limited access. I can't let you in. Are you looking for anyone in particular?*

*Intruder: Yes, I'm looking for Mr. Cameron.*

*Guard: Please wait here. I'll find him for you.*

Another way to protect a host is to protect it completely from “direct” contact with untrusted execution units. Whenever an untrusted execution unit requests the execution of an operation, a trusted party, *the arbitrator*, is called in to carry out

the operation for the execution unit. By restricting the kind of operations visible to the execution unit, and by examining the client's run-time state, the arbitrator can carefully block out unsafe operations. The cost of such flexibility is usually run-time performance penalty.

Arbitration can be used to enforce both memory protection and access control. An *abstract interpreter* is often used to enforce memory protection. *Interposition* is frequently used for enforcing access control. We will examine each of them in turn.

#### 4.4.1 Memory Protection by Abstract Interpreter

Using an abstract interpreter has been a very popular way of implementing safe and portable computation. Mobile code languages like Java [50], Safe Tcl [84], Scheme48 [91], and Telescript [125], JavaScript [80], all involve the interpretation of some source or intermediate languages. The interpreter approach can achieve memory protection in two ways:

1. **Restricting expressiveness:** A safe intermediate representation can be defined for mobile code units. Due to language restrictions, certain unsafe operations cannot be expressed, while others can be statically checked for. Take the JVM byte-code representation [65] as example. Privileged native instructions cannot be expressed; there is no pointer arithmetic; the language is strictly typed; interactions with the host's resources are performed through public application programming interface (API) [17]. As a result, memory interference can be avoided.
2. **Dynamic checking:** The execution unit interacts with the host's CPU only through the arbitration of the interpreter. Consequently, the interpreter can screen out all potentially dangerous moves by run-time checking. For example, the JVM checks against null pointer dereferencing, out-of-bound array access, and illegal type-cast [65].

#### 4.4.2 Evaluation of Abstract Interpretator

- With interpretation, the host is completely isolated from the execution unit, thereby realizing the principle of complete mediation (section 3.2.1).
- The complete redefinition of the underlying computing model offers the flexibility of raising the level of abstraction and introducing run-time security checks. With a more abstract representation, mobile code units are easier to verify, both statically and dynamically.
- Program representation is not dependent on the computing platform of the host, making mobile code programs more easily portable among heterogeneous platforms.
- Interpretation is usually a major bottleneck for performance. The extra level of indirection introduces significant interference. Languages like Java, Omniware, and Juice now offer JIT compilation [127, 66, 43] as a means for speeding things up.
- It is interesting to know if, by appropriately restricting the expressiveness of an interpreted language, one may enforce higher level security concerns other than

memory protection. For example, in PLAN, a functional, active packet language [54], programs are guaranteed to terminate by disallowing recursive function call.

#### 4.4.3 Access Control by Interposition

Interposition is the insertion of trusted arbitration code, usually called a *reference monitor* [96], between a protected service and the entry point of the service. In a mobile code system with a mature memory protection mechanism, execution units usually access host resources via a set interface. All attempts to access the protected resources must then pass through the interface and are therefore subject to the monitoring of the trusted arbitration code before reaching the target service. Access control policies may be programmed into the arbitration code, which can flexibly screen out inappropriate access to the service. Interposition usually requires no modification to the host's hardware or operating system. So, it has become a very popular mechanism for the layered protection of system services. In this section, we will survey several implementations of interposition.

**Janus and Arbitrated System Calls.** Janus [45] itself is not a mobile code system. However, it is an application wrapper designed to protect the host against untrusted mobile code computing environment. The intuition behind its design is that an untrusted process cannot do much harm if its access to the underlying operating system is appropriately restricted. Using the process tracing facilities and the `/proc` virtual file system in Solaris, Janus creates a user-level sandbox that monitors all system calls made by an untrusted process. Since legacy computing environments which have unreliable protection mechanisms (e.g. an old version of `ghostview` [16], `sh`, or a buggy, Java-enabled web browser) can be executed inside the Janus sandbox, the Janus monitor can effectively block out unsafe system access initiated by the execution units running inside the legacy computing environment. Users may even supply their own policy module to specify which system calls to allow, which one to deny, and for which a function must be called to determine what to do.

Janus represents a very practical solution to a very practical problem. It does not require modification to the kernel and the computing environment, and it effectively protects the host from any unreliable computing environment. However, even disregarding its platform dependent nature, Janus does not satisfactorily address the layered protection problem. Firstly, Janus does not allow the computing environment to define a different protection domain for each execution unit. Secondly, the kind of security policy it may express is limited due to its ignorance of the semantics of the computing environment. For instance, when a JVM is running inside a Janus sandbox, the policy modules of Janus have no way of figuring out the internal state of the JVM, and must make their access control decision independent of the JVM's state. In general, layered protection is adequately addressed only when introspection is a built-in feature of the computing environment instead of being a retrofitted patch of the operating system.

**Java and the Security Manager** Java implements interposition by using two mechanisms — the *security manager* and *stack inspection*<sup>16</sup>. All access to operating system services are isolated in the standard Java API [17]. Whenever a service routine is invoked, the API transfers control to a corresponding monitor method of the global security manager object. The monitor method will inspect the Java run-time stack to determine if the call is safe. If the monitor method disallows the access, an exception will be thrown. Otherwise, control is returned to the service routine, carrying out the original request. The security authority may override these monitor methods of the security manager class in order to customize the security policy of the JVM.

The Java security model allows one to define intricate security policies. Stack inspection allows the security manager to decide with fine control what access will be granted. The drawbacks of this approach are as follows. Firstly, the security manager needs to implement complex stack inspection logic to differentiate between accesses initiated by different execution units. From a software engineering point of view, the construction and maintenance of this logic is both difficult and prone to error. Secondly, a procedural definition of security policy is hard to understand. A popular solution is to implement traditional access control lists in the arbitration code (as in Java [58, 47, 46, 48, 49] and Agent Tcl [51]). Recently, Netscape has attempted to extend the Java stack inspection mechanism by providing stack annotation which simplifies the logic for access right checking [122]. This extended version of stack inspection is later proven by Wallach and Felten [123] to be equivalent to formal deduction in ABLP logic [2].

**Safe-Tcl and Name Resolution Control** Safe-Tcl [84] is a security-aware extension of the popular Tcl scripting language [83, 124]. Protection is achieved by three mechanisms — *safe interpreters*, *aliases*, and *hidden commands*. Tcl is a command-based language, similar to other shell scripting languages. Access to operating system facilities are provided through a set of commands. Safe-Tcl defines a *padded cell security model*, in which every execution unit is executed in its own interpreter. All system services are available in a trusted, master interpreter. When an untrusted script is executed, it is sandboxed in a separate, untrusted, safe interpreter. A safe interpreter acts like a separate name space. Privileged commands can be *hidden* in the safe interpreter, thus blocking untrusted script from unauthorized access to system resources. Also, to obtain finer-grained control, a command may be *aliased*. Specifically, the name of a privileged command in the safe interpreter may be “overshadowed” by a trusted arbitration routine in the master interpreter. The arbitration routine decides at run-time if the access is granted. If the access is permitted, it delegates the original call to the overshadowed command in the master interpreter.

The padded cell model represents the most general form of interposition — *name resolution control*. In essence, the name resolution control is composed of two component mechanisms. Firstly, granting of capabilities is realized by *link-time visibility control*. The notion of a safe interpreter, which is essentially a name space, coincides with a protection domain. A privileged service can be accessed only if it can be *named* in the safe interpreter. The assignment of each script to a separate name space, plus

---

<sup>16</sup>In [122], Wallach *et al* refer to the mechanism as *stack introspection*. Later on, they call it *stack inspection* in [123].

the possibility of name hiding, allow one to easily tailor a different access policy for each script. This makes the implementation of security policy a lot more tractable. Secondly, fine-grained *run-time accessibility checks* can be programmed into aliased arbitration routines. Here, accessibility is not controlled by visibility, but instead by dynamic checking of the possession of rights.

Scheme48 [91] is another early mobile code system that uses name resolution control as its primary protection mechanism. In Scheme, a procedure is a function closure, containing a lambda expression and a binding environment. When a procedure is applied, the only objects that are visible inside the lambda expression are the actual arguments and the values of the names in the lexical environment. Scheme48 allows programs to construct arbitrary binding environments, and then execute untrusted code inside these carefully-crafted environment. During the course of constructing such environments, privileged procedure names can be made invisible or be redefined to refer to arbitration routines.

Wallach *et al* [122] describe a way to implement name resolution control in the context of Java. In Java, a name space coincides with a classloader. A class name in one classloader represents a different class than another class with the same name in a different classloader. The classloader was originally conceived for name space partitioning so that there will be no name conflict among separate execution units. Taking advantage of this design, one may create a subclass of the standard classloader class, in which all requests for name resolution is monitored. As a result, if a privileged name is to be hidden, the classloader can throw an exception when the name is resolved. Aliasing can be simulated by resolving the names of privileged classes to arbitration classes.

The extensible operating system SPIN [11, 104] also models protection domains by name spaces. All extension code in SPIN is written in the type-safe language Modula-3. Capabilities are directly modeled as pointers. Therefore, if a name is well-typed in a code unit, then the resource or service it refers to will be accessible. As such, typing provides a means of expressing *conditional visibility* of a symbol. Fine-grained protection is achieved by allowing users to manipulate name spaces. Name spaces can be created dynamically, and code units are executed within the confine of that name space, thus restricting its capabilities. An interesting feature is that name spaces can be extended by the **Combine** operation, which creates a union of two name spaces. This compares with the more flexible name space extension primitive **import** mentioned in section 2.3. In general, a system that uses name resolution control for protection needs ways to construct and extend name spaces.

One potential problem with modeling protection domains using name spaces is that there is no way of revoking capability. The J-Kernel, proposed by Hawblitzel *et al* [52], is a Java security kernel that provides capability revocation mechanism within a name-space-as-protection-domain framework.

#### 4.4.4 Evaluation of Interposition

- Name spaces provide a very clean model for implementing protection domains in a programming language level. By allowing the creation and extension of name spaces, the challenge of layered protection is adequately addressed.
- Reference monitors usually introduce run-time overhead to the execution units if



they are invoked very often.

## 5 Perspective

This survey leads to the identification of some future research directions:

1. Research in the host security of mobile code system should be judged on how well they address the challenges of anonymous trust, layered protection, and implicit acquisition.
2. The discretion approach can be made more reliable if the semantics of signatures specify program-analytic constraints and properties. Therefore, formalization of security policies into program analytic terms is desired.
3. Static methods like verification and transformation appear to be an ideal candidate for addressing the challenge of anonymous trust. Further development of such approaches to deal with high level access control policy is desired.
4. Although researches on secure information flow control has begun to address confidentiality issues, development of techniques for using static analysis to deal with integrity issues is less mature.
5. The verification approach introduces many software engineering challenges like minimization of link-time cost, link-safety, reduced disclosure of intellectual properties, configuration management, interoperability, modularity of the verification architecture, and others. Exploration of novel verification protocols to address such challenges will remain a very important research topic.
6. Name resolution control is found to be a desirable programming language analogue for protection domain. It offers a very clean abstraction for tackling the problem of layered protection. Formalization of name resolution control into a well-understood programming language construct is an interesting line of research.

## References

- [1] Martin Abadi. Secrecy by typing in security protocols. In *Proceedings of the Third International Conference on Theoretical Aspects of Computer Science (TACS'97)*, Lecture Notes in Computer Science, pages 611–638. Springer-Verlag, September 1997. Also available at <ftp://ftp.digital.com/pub/DEC/SRC/publications/ma/types-sv.ps>.
- [2] Martin Abadi, Michael Burrows, Butler Lampson, and Gordon D. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993. Also available at <http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-070.html>.
- [3] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.
- [4] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International*

*Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1998.

- [5] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proceedings of ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96)*, pages 127–136, May 1996. Also available at <http://www.cs.cmu.edu/afs/cs.cmu.edu/user/ali/www/pldi96-omniware.ps>.
- [6] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, Jan 1980.
- [7] Dirk Balfanz and Ed Felten. A Java filter. Technical Report 567-97, Department of Computer Science, Princeton University, September 1997. Available at <http://www.cs.princeton.edu/sip/pub/javafilter.html>.
- [8] Jean-Pierre Banâtre, Ciarán Bryce, and Daniel Le Métayer. Compile-time detection of information flow in sequential programs. In *Proceedings of the Third European Symposium on Research in Computer Security*, volume 875 of *Lecture Notes in Computer Science*, pages 55–73, Brighton, United Kingdom, Nov 1994. Also available at <ftp://ftp.irisa.fr/local/lande/dlm-esorics94.ps.Z>.
- [9] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations and model. *MITRE Report*, MTR 2547 v2, Nov 1973.
- [10] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, David Becker, Marc Ficuzynski, and Emin Gun Sirer. Protection is a software issue. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 62–65, Orcas Island, WA., 1995. Also available at <http://www.cs.washington.edu/research/projects/spin/www/papers/HotOS95/hotos95.ps>.
- [11] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Ficuzynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 267–284, 1995. Also available at <http://www.cs.washington.edu/research/projects/spin/www/papers/SOSP95/sosp95.ps>.
- [12] K. Biba. Integrity considerations for secure computer systems. Technical Report 76–372, U. S. Air Force Electronic Systems Division, 1977.
- [13] Nathaniel S. Borenstein. EMail with a mind of its own: The Safe-Tcl language for enabled mail. In *IFIP WG 6.5 Conference*, Barcelona, North Holland, Amsterdam, May 1994. Available as part of <ftp://ftp.fv.com/pub/code/other/safe-tcl.tar.Z>.
- [14] Luca Cardelli. Type systems. In Allen B. Tucker Jr., editor, *The Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press, 1997. Also available at <http://research.microsoft.com/research/cambridge/luca/Papers/TypeSystems.ps>.
- [15] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing distributed applications with mobile code paradigms. In *Proceedings of the 19th*

- International Conference on Software Engineering (ICSE'97)*, 1997. Also available at <http://www.polito.it/~picco/papers/icse97.ps.gz>.
- [16] CERT\* Coordination Center. Ghostscript vulnerability. CERT\* Advisory CA-95.10, August 1995. Available at <http://www.cert.org/advisories/CA-95.10.ghostscript.html>.
  - [17] Patrick Chan and Rosanna Lee. *The Java Class Libraries: An Annotated Reference*. The Java Series. Addison-Wesley, 1996.
  - [18] Phil Inje Chang. Inside the Java Web server: An overview of Java Web Server 1.0, Java servlets, and the JavaServer architecture, 1997. Available at <http://java.sun.com/features/1997/aug/jws1.html>.
  - [19] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4), Nov 1994. Also available at <ftp://ftp.cs.washington.edu/tr/1993/04/UW-CSE-93-04-02.PS.Z>.
  - [20] Thomas M. Chen and Alden W. Jackson, editors. *Special Issue on Active and Programmable Networks*. IEEE Network Magazine. IEEE, July/August 1998.
  - [21] David Chess, Benjamin Grosz, Colin Harrison, David Levine, Colin Parris, and Gene Tsudik. Itinerant agents for mobile computing. Technical Report RC 20010, IBM Research Division, 1995. Also available at <http://www.research.ibm.com/massdist/rc20010.ps>.
  - [22] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile agents: Are they a good idea? In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 25–45. Springer-Verlag, 1997. Also available at <http://www.research.ibm.com/massdist/mobag.ps>.
  - [23] David M. Chess. Security issues in mobile code systems. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
  - [24] Chrysalis-ITS, Inc. Chrysalis-ITS home page. Available at <http://www.chrysalis-its.com>.
  - [25] D. Clark and D. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 184–194, 1987.
  - [26] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 2nd edition, 1996.
  - [27] Patrick Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 15, pages 841–993. MIT Press, 1990.
  - [28] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analyzing mobile code languages. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 93–110. Springer-Verlag, 1997. Also available at <http://www.polito.it/~picco/papers/ecoop96.ps>.

- [29] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
- [30] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From Hot-Java to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1996.
- [31] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
- [32] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
- [33] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.
- [34] P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. In *Information Processing (Proceedings of the IFIP Congress)*, pages 320–326, 1971.
- [35] Premkumar T. Devanbu, Philip W. L. Fong, and Stuart G. Stubblebine. Techniques for trusted software engineering. In *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan, 1998. Available at <http://seclab.cs.ucdavis.edu/~devanbu/icse98.ps>.
- [36] Premkumar T. Devanbu and Stuart G. Stubblebine. Research directions for automated software verification: Using trusted hardware. In *Proceedings of the Twelve IEEE International Conference on Automated Software Engineering (ASE'97)*, Lake Tahoe, Nevada, November 1997. Available at <http://seclab.cs.ucdavis.edu/~devanbu/ase97.ps>.
- [37] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, Dec 1995. Also available at <http://www.pdos.lcs.mit.edu/papers/exokernel-sosp95.ps>.
- [38] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Authentication and state appraisal. In *Proceedings of the Fourth European Symposium on Research in Computer Security (ESORICS'96)*, volume 1146 of *Lecture Notes in Computer Science*, pages 118–130, Rome, Italy, September 1996. Springer-Verlag.
- [39] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Issues and requirements. In *Proceedings of the 19th National Information Systems Security Conference*, pages 591–597, 1996. Also available at <http://csrc.nist.gov/nissc/1996/papers/NISSC96/paper033/SWARUP96.PDF>.
- [40] Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach. Web spoofing: An internet con game. In *Proceedings of the 20th National Information Systems Security Conference*, Baltimore, Maryland, October 1997. Also available at <http://www.cs.princeton.edu/sip/pub/spoofing.html>.
- [41] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

- [42] Philip W. L. Fong and Robert D. Cameron. Proof linking: An architecture for modular verification of dynamically-linked mobile code. In *Proceedings of the 6th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-6)*, Orlando, Florida, November 1998. Also available at <http://www.cs.sfu.ca/~pwfong/personal/Pub/fse98.ps>.
- [43] Michael Franz and Thomas Kistler. Juice. Available at <http://www.ics.uci.edu/~juice/>.
- [44] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, San Antonio, Texas, January 1999. Also available at <http://simon.cs.cornell.edu/home/jgm/papers/mtal.pdf>.
- [45] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, July 1996. Also available at <http://www.cs.berkeley.edu/~daw/janus-usenix96.ps>.
- [46] Li Gong. Java security: Present and near future. *IEEE Micro*, 17(3):14–19, May/June 1997. Also available at <http://java.sun.com/people/gong/papers/IeeeMicro97.pdf>.
- [47] Li Gong. New security architectural directions for Java (extended abstract). In *Proceedings of IEEE COMPCON*, pages 97–102, San Jose, CA., Feb 1997. Also available at <http://java.sun.com/people/gong/papers/Ieee-compcon97.ps.gz>.
- [48] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA., Dec 1997. Also available at <http://java.sun.com/people/gong/papers/jdk12arch.ps.gz>.
- [49] Li Gong and Roland Schemers. Implementing protection domains in the Java Development Kit 1.2. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, San Diego, CA., March 1998. Also available at <http://java.sun.com/people/gong/papers/jdk12impl.ps.gz>.
- [50] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996. Also available at <http://www.javasoft.com/docs/books/jls/index.html>.
- [51] Robert S. Gray. Agent Tcl: A flexible and secure mobile agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop*, pages 9–23, Monterey, CA., July 1996. Also available at <http://www.cs.dartmouth.edu/~agent/papers/tcl96.ps.Z>.
- [52] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA., June 1998. Also available at <http://www.cs.cornell.edu/slk/papers/usenix98.ps>.

- [53] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, San Diego, CA., January 1998. Available at <http://cm.bell-labs.com/who/riecke/bib.html>.
- [54] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A programming language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages (ICFP'98)*, pages 86–93, 1998. Also available at <http://www.cis.upenn.edu/~switchware/papers/plan.ps>.
- [55] IBM. IBM 4758 PCI cryptographic coprocessor. Available at <http://www.ibm.com/Security/cryptocards/>.
- [56] Trent Jaeger, Atul Prakash, and Aviel D. Rubin. A system architecture for flexible control of downloaded executable content. In *Proceedings of the Fifth International Workshop on Object-Oriented in Operating Systems (IWOOOS'96)*, pages 14–18. IEEE, 1996.
- [57] JavaSoft. Java security: Chronology of security-related bugs. Available at <http://www.javasoft.com/sfaq/chronology.html>.
- [58] JavaSoft. Security and signed applets. Available at <http://www.javasoft.com/products/jdk/1.1/docs/guide/security/>.
- [59] Kazuhiko Kato. Safe and secure execution mechanisms for mobile objects. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 201–211. Springer-Verlag, 1997.
- [60] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World*. Prentice Hall, 1995.
- [61] Frederick C. Knabe. *Language Support for Mobile Agents*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA, December 1995.
- [62] Mark D. LaDue. A collection of increasingly hostile applets. Available at <http://www.cyber.com/mirror/mladue/HostileApplets.html>.
- [63] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. The Java Series. Addison-Wesley, 1997.
- [64] Stanley Letovsky and Elliot Soloway. Delocalized plans and program comprehension. *IEEE Software*, pages 41–49, May 1986.
- [65] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1997. Also available at <http://www.javasoft.com/docs/books/vmspec/index.html>.
- [66] S. Lucco, O. Sharp, and R. Wahbe. Omniware: A universal substrate for web programming. In *Proceedings of the Fourth International World Wide Web Conference*, December 1995. Also available at <http://www.cs.utah.edu/~ampsem/omniware.html>.

- [67] Terry Mayfield, J. Eric Roskos, Stephen R. Welke, and John M. Boone. Integrity in automated information systems. Technical Report C-TR-79-91, National Computer Security Center, September 1991. Also available at <http://www.radium.ncsc.mil/tpep/library/rainbow/C-TR-79-91.ps>.
- [68] Gary McGraw and Ed Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley and Sons, 1997.
- [69] Ir. Marc Meurrens. Java code engineering & reverse engineering. Available at <http://Meurrens.ML.org/ip-Links/Java/codeEngineering/>.
- [70] Masaaki Mizuno. A least fixed point approach to inter-procedural information flow control. In *Proceedings of the Twelve National Computer Security Conference*, pages 558–570, Baltimore, MD., Oct 1989.
- [71] Masaaki Mizuno and Arthur E. Oldehoeft. Information flow control in a distributed object-oriented system with statically bound object variables. In *Proceedings of the Tenth National Computer Security Conference*, pages 56–67, Sept 1987.
- [72] Masaaki Mizuno and David Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4:727–754, 1992.
- [73] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Workshop on Types in Compilation*, Kyoto, Japan, March 1998. Also available at <http://simon.cs.cornell.edu/home/jgm/papers/fullstal.ps>.
- [74] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language (extended version). Technical Report TR97-1651, Cornell University, November 1997. Also available at <http://simon.cs.cornell.edu/home/jgm/papers/tal-tr.ps>.
- [75] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 85–97, San Diego, CA., January 1998. Also available at <http://simon.cs.cornell.edu/home/jgm/papers/tal.ps>.
- [76] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997. Also available at <http://www.cs.cmu.edu/~necula/pop197.ps.gz>.
- [77] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI'96)*, Seattle, WA., October 1996. Also available at <http://www.cs.cmu.edu/~necula/osdi96.ps.gz>.
- [78] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 333–344, Montreal, Canada, Nov 1998. Also available at <http://www.cs.cmu.edu/~necula/pldi98.ps.gz>.

- [79] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998. Also available at <http://www.cs.cmu.edu/~necula/lncs98.ps.gz>.
- [80] Netscape Communications Co. JavaScript developer central, 1998. Available at <http://developer.netscape.com/tech/javascript/>.
- [81] Peter Ørbæk. Can you trust your data? In P. D. Mosses, M. I. Schwartzbach, and M. Nielsen, editors, *Proceedings of the TAPSOFT/FASE'95 Conference*, volume 915 of *Lecture Notes in Computer Science*, pages 575–590, Aarhus, Denmark, May 1995. Springer-Verlag. Also available at <ftp://ftp.daimi.aau.dk/pub/empl/poe/trust.ps.gz>.
- [82] Peter Ørbæk and Jens Palsberg. Trust in the  $\lambda$ -calculus. *Journal of Functional Programming*, 7(4), 1997. Also available at <ftp://ftp.daimi.aau.dk/pub/empl/poe/lambda-trust-full.ps.gz>.
- [83] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
- [84] John K. Ousterhout, Jacob Y. Levy, and Brent B. Welch. The Safe-Tcl security model. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998. Also available as Sun Microsystems Laboratories technical report TR-97-60, at <http://research.sun.com/technical-reports/1997/abstract-60.html>.
- [85] Jens Palsberg and Peter Ørbæk. Trust in the  $\lambda$ -calculus. In Alan Mycroft, editor, *SAS'95: Static Analysis*, volume 983 of *Lecture Notes in Computer Science*, pages 314–330, Glasgow, Sept 1995. Springer-Verlag. Also available at <ftp://ftp.daimi.aau.dk/pub/empl/poe/lambda-trust.ps.gz>.
- [86] Flavio De Paoli, Andre L. Dos Santos, and Richard A. Kemmerer. Web browsers and security. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
- [87] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2nd edition, 1996.
- [88] Charles P. Fleeger. *Security in Computing*. Prentice Hall, 2nd edition, 1997.
- [89] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.
- [90] Christian Queinnec and David De Roure. Sharing code through first-class environments. In *Proceedings of the 1996 International Conference on Functional Programming*, May 1996.
- [91] Jonathan A. Rees. A security kernel based on the lambda-calculus. A. I. Memo 1564, MIT, 1996. Also available at <ftp://publications.ai.mit.edu/ai-publications/1500-1999/AIM-1564.ps.Z>.
- [92] Stuart Ritchie. Systems programming in java. *IEEE Micro*, pages 30–35, May/June 1997.
- [93] Eva Rose and Kristoffer Høgsbro Rose. Lightweight bytecode verification. In *The OOPSLA'98 Workshop on Formal Underpinnings of Java*, Vancouver, BC,



- Canada, November 1998. Available at <http://www-dse.doc.ic.ac.uk/~sue/oopsla/rose.f.ps>.
- [94] François Rouaix. A web navigator with applets in Caml. In *Proceedings of the 5th WWW Conference*, Paris, May 1996. Also available at <http://pauillac.inria.fr/~rouaix/mmm/papers/mmm.ps.gz>.
  - [95] Spencer Rugaber, Kurt Stirewalt, and Linda W. Wills. The interleaving problem in program understanding. In *Proceedings of the 2nd Working Conference on Reverse Engineering*, pages 166–175, Toronto, Ontario, July 1995. Also available at <ftp://ftp.ee.gatech.edu/pub/reveng/wcre95-interleaving.ps>.
  - [96] John Rushby and Brian Randell. A distributed secure system. *IEEE Computer*, pages 55–67, July 1983.
  - [97] Jerome H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.
  - [98] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept 1975.
  - [99] Ravi S. Sandhu. On five definitions of data integrity. In *Proceedings of the IFIP WG11.3 Workshop on Database Security*, Lake Guntersville, Alabama, Sept 1993. Also available at [http://www.list.gmu.edu/confrnc/ifip/ps\\_ver/i93int.ps](http://www.list.gmu.edu/confrnc/ifip/ps_ver/i93int.ps).
  - [100] David A. Schmidt. *The Structure of Typed Programming Language*. MIT Press, 1994.
  - [101] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, 1996.
  - [102] Margo Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. An introduction to the architecture of the vino kernel. Technical Report 34-94, Harvard Computer Center for Research in Computing Technology, 1994. Also available at <ftp://das-ftp.harvard.edu/techreports/tr-34-94.ps.gz>.
  - [103] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, 4th edition, 1994.
  - [104] Emin Gun Sirer, Marc Fiuczynski, Przemyslaw Pardyak, and Brian Bershad. Safe dynamic linking in an extensible operating system. In *Proceedings of the Workshop on Compiler Support for System Software*, Feb 1996. Also available at <http://www.cs.washington.edu/research/projects/spin/www/papers/WCS/domain.ps>.
  - [105] Christopher Small and Margo Seltzer. A comparison of OS extension technologies. In *Proceedings of the 1996 USENIX Conference*, 1996. Also available at <http://www.eecs.harvard.edu/~vino/vino/papers/bakeoff.ps>.
  - [106] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 355–364, San Diego, CA., January 1998. Also available at <http://www.cs.nps.navy.mil/research/languages/papers/atsc/pop198.ps.Z>.

- [107] Spyrus, Inc. Spyrus home page. Available at <http://www.spyrus.com>.
- [108] James W. Stamos and David K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.
- [109] Ed Taft and Jeff Walden. *PostScript Language Reference Manual*. Addison-Wesley, 2nd edition, 1990.
- [110] Joseph Tardo and Luis Valente. Mobile agent security and Telescript. In *Proceedings of the COMPCON'96*, pages 58–63. IEEE, Feb 1996.
- [111] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.
- [112] The Active Group. The Active Group home page. Available at <http://www.activex.org>.
- [113] The Princeton Secure Internet Programming Group. Secure internet programming. Available at <http://www.cs.princeton.edu/sip/>.
- [114] The Princeton Secure Internet Programming Team. Security trade-offs: Java vs ActiveX. Available at <http://www.cs.princeton.edu/sip/java-vs-activex.html>.
- [115] The Washington Kimera Group. Kimera: A Java system architecture. Available at <http://kimera.cs.washington.edu/>.
- [116] Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, September 1997. Also available at <ftp://ftp.inria.fr/INRIA/publication/RR/RR-3134.ps.gz>.
- [117] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings of the Tenth IEEE Computer Security Foundations Workshop*, pages 156–168, Rockport, MA., June 1997. Also available at <http://www.cs.nps.navy.mil/research/languages/papers/atsc/csfw97.ps.Z>.
- [118] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of the Seventh International Joint Conference on the Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621, April 1997. Also available at <http://www.cs.nps.navy.mil/research/languages/papers/atsc/tapsoft97.ps.Z>.
- [119] Dennis Volpano and Geoffrey Smith. Language issues in mobile program security. In *Mobile Agents and Security*, Lecture Notes in Computer Science. Springer-Verlag, 1998. To appear. Also available at <http://www.cs.nps.navy.mil/research/languages/papers/atsc/lncs98.ps.Z>.
- [120] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996. Also available at <http://www.cs.nps.navy.mil/research/languages/papers/atsc/jcs.ps.Z>.

- [121] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, 1993. Also available at <http://http.cs.berkeley.edu/~tea/sfi.ps>.
- [122] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. Technical Report TR 546-97, Department of Computer Science, Princeton University, April 1997. Also available at <http://www.cs.princeton.edu/sip/pub/sosp97.html>.
- [123] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proceedings of 1998 IEEE Symposium on Security and Privacy*, 1998. Also available at <http://www.cs.princeton.edu/sip/pub/oakland98.html>.
- [124] Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall, 1995.
- [125] James E. White. Mobile agents. In Jeffrey Bradshaw, editor, *Software Agents*, chapter 19, pages 437–472. AAAI Press/MIT Press, 1996.
- [126] Frank Yellin. Low level security in Java. In *World-Wide Web Conference*, Boston, MA., December 1995. Also available at <http://www.javasoft.com/sfaq/verifier.html>.
- [127] Frank Yellin. *The Java Native Code API*. JavaSoft, 1997. Available at [http://www.javasoft.com/docs/jit\\_interface.html](http://www.javasoft.com/docs/jit_interface.html).