Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

J. Parallel Distrib. Comput. 64 (2004) 191–200

Journal of Parallel and Distributed Computing

http://www.elsevier.com/locate/jpdc

Protecting network users in mobile code systems[☆]

Shiuh-Pyng Shieh* and Wen-Her Yang

Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu 30010, Taiwan, ROC

Received 14 January 2000

Abstract

Conventional access control mechanisms are rather insensitive to occurrences of context-dependent illegal accesses. Insensitivity to context-dependent accesses may lead to failure to protect network users and resources. Context-dependent illegal accesses resulting from data and privilege flows in open networks cannot be prevented by either authentication or access control mechanisms since unauthorized access need not be attempted. In this paper we present a protection model which tracks data and privilege flows in mobile code systems. It can uniformly define various types of illegal access patterns and has the advantage of preventing context-dependent illegal accesses such as those caused by inadvertent execution of remote mobile code containing viruses or Trojan Horses. The proposed flow control model is expected to complement the conventional model for access control.
© 2003 Elsevier Inc. All rights reserved.

Keywords: Mobile code; Remote execution; Java; Internet security; World Wide Web; Access control; Flow control

1. Introduction

With the rapid growth of the Internet, many services are provided to help network users to access remote data and code. The development of World Wide Web (WWW) combines many traditional services and allows users to navigate the entire Internet using a single Web browser [Bern94]. As users are expecting more and more new features, many extensions to the standard World Wide Web interface are introduced. An extension to WWW for distributing applications involves "mobile code"—code that can be transmitted across the network and executed on the other end. The Java^{TM} language is a simple, object-oriented, portable, robust language that supports mobile codes [Flanagan96,Gosling95,Gosling96,Hot96]. Java augments the present WWW capabilities by dynamically downloading the mobile code fragments, called applets, and running these code fragments locally. Since the mobile codes are transmitted across insecure networks from possibly untrusted sources and executed in the local browser, it raises serious security issues [Dean96]. Therefore, in the design of mobile applications on the Internet, the security problem is considered to be an important one. No one wants to bring across any piece of code if there is a possibility that executing the code could (1) damage any hardware, software, or information on the host machine, (2) pass unauthorized information to anyone [Yellin95].

Both authentication and access control can help protect network users in the World Wide Web [Shieh96,Shieh97a,Shieh99]. However, access control mechanisms are rather insensitive to occurrences of context-dependent illegal accesses. Context-dependent illegal accesses may arise from a sequence of authorized executions that exploit security weaknesses of mobile code systems. For example, inadvertent execution of a mobile code containing Trojan Horse, modification of local users' sensitive object, and disclosure of sensitive data to remote users. Insensitivity to context-dependent accesses may lead to failure to protect network users and resources. Context-dependent illegal accesses resulting from data and privilege flows in open networks cannot be prevented by either authentication or access control mechanisms because unauthorized direct access need not be attempted. Here, data and privilege flows are generally caused by primitive system operations, for example, users access files and execute programs. The

[☆]This work was supported in part by the National Science Council of Taiwan, ROC, under Contract NSC-85-2213-E-009-032. Java^{TM} is a registered trademark of Sun Microsystems. UNIX^{TM} is a registered trademark of AT&T.

*Corresponding author. Fax: +886-3572-4176.
E-mail address: ssp@csie.nctu.edu.tw (S.-P. Shieh).

formal definition of data and privilege flows is given in Section 3. Since access control mechanisms do not track data and privilege flows between subjects (e.g., users and processes) and objects (e.g., files and memory segments), many illegal accesses cannot be prevented when the access patterns include sequences of accesses in which both the size of the subject and object sets and the length of the sequence itself vary in time. This limitation is shared by access control methods that ignore the meaning and significance of event sequences in defining illegal access patterns. After-the-fact some flow models [Denning76,Denning77,Landwehr81] and audit analysis [Shieh97b] have been used to detect this type of intrusions. However, all of them are unable to provide immediate, real-time protection.

The current Java security manager protects the users' system resources by using access control list (ACL) [Yellin95]. The users can configure each ACL in the browser to specify what resource access are permitted or denied. To ensure the security of the local user and host, the Web browser has to limit the access to system resources such as the file system, the CPU, the network, and the graphics display. Otherwise, a malicious hacker who wrote the mobile code could read or write the user's personal files, send anonymous mails to other hosts, introduce viruses, or crack the local host [Cert96a,-Cert96b]. Conventional access control mechanisms limit the access of a process to a file. This will not be effective in mobile code systems where a mobile code may contain Trojan Horses. For security reasons, an unsigned Java applet in Netscape browser is currently not permitted to access files, and it is only allowed to communicate with the server from which the applet is downloaded. However, these restrictions limit the capability of mobile code systems and the resource sharing among network users. Under these restrictions, many jobs cannot be accomplished using Java applets, which is undesirable. Therefore, a new scheme is needed for network users in mobile code systems to protect themselves, and at the same time to share resources.

In this paper, we propose a protection model for mobile code applications, which captures the dynamic flows of data and privileges, thereby enabling the definition and prevention of specific illegal access patterns. The operations on the subject/object are permitted if the flows are allowed. This model can address context-dependent illegal accesses resulting from unintended execution of Trojan Horses, and inadvertent propagation of data and code fragments. It monitors privilege and data flows in a mobile code network system, and is expected to complement, not replace, current access control mechanisms.

This paper is organized as follows. In Section 2, we present illegal data and privilege flows resulting from sequences of actions in mobile code systems that help motivate the flow control approach to network user protection. Section 3 contains a succinct definition of the data and privilege flows and their salient properties. In Section 4, we will formally describe the flow control model. Based on the model, various types of illegal access patterns are uniformly defined and prevented in Section 5. Finally, we give the conclusions in Section 6.

2. Security problems in mobile code systems

The problems of illegal data and privilege flows arise from complex interactions between the pieces of mobile codes, objects (e.g., files and memory segments) and network users, such as illegal write over sensitive files while executing a mobile code that contains Trojan Horses, or propagation and disclosure of sensitive data. These problems affect the mobile code systems employing access controls, and may be caused by both privileged, administrative users and unprivileged, casual users. In this section, we present three types of security problems leading to context-dependent illegal accesses, and which suggest the use of a data and privilege flows approach to the protection of network users. The examples are fairly typical in mobile code systems.

2.1. *Modification and disclosure of client's sensitive data*

The following example illustrates the effect of the execution of a mobile code by an unsuspecting user. A hacker deliberately leaves a mobile code on a Web server to attract the attention of network users. An unsuspecting user may download and execute the mobile code. While executing the code, both the mobile code and the user who wrote the code can acquire all the unsuspecting user's privilege, thereby controlling all his activities and allowing the hacker to (1) read the unsuspecting user's sensitive objects and send them back to the hacker (see Fig. 1); (2) write over the unsuspecting user's sensitive objects (see Fig. 2). For example in the Microsoft browser (Internet Explorer), if the policy of Java security is not properly configured, an unsigned Java applet may acquire the privilege to access file systems. Consequently, a hacker is able to disclose or modify the sensitive files through a malicious Java applet. The unsuspecting user can set access control rules to forbid the browser to download mobile codes from untrusted

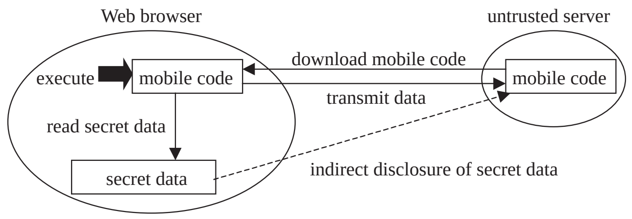


Fig. 1. Indirect disclosure of secret data.

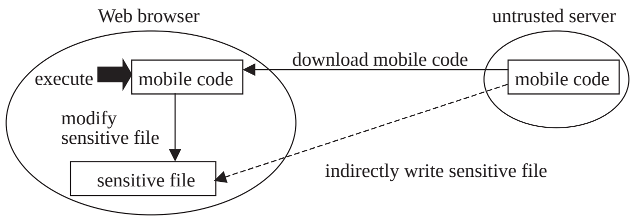


Fig. 2. Indirectly write over a sensitive file through a mobile code.

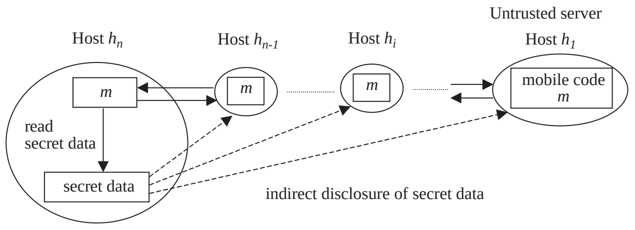


Fig. 3. Indirectly cascading disclosure of secret data.

servers; however, the problem still exists. For example in Fig. 3, host h_n is not trusted by host h_1, the mobile code m in h_1 may travel to other hosts which are trusted by h_n. If the user in h_n downloads and executes the mobile code m from these trusted hosts, then m is able to disclose user's secret data to all the hosts where m ever traveled. In this context, these seemingly normal actions of the mobile code represent illegal accesses and cannot be prevented by access control mechanisms.

Although it is possible in JDK (Java Development Kit) and Netscape to deny an applet's request to open a TCP/IP connection back to the server from which it was loaded, the confinement problem [Lampson73,Lipner75,Saltzer75] still exists in WWW that secret data may flow outside a protected host (that is, secret data of a network user may be disclosed to a remote Web server). For example, a user executes an applet which is allowed to read secret system data, e.g., the user password file, and to write some temporary files in the /tmp directory, but is forbidden making any network connections back to the server where it was downloaded. The applet may write data in the public directory /tmp and leave it there. Later, the user executes another applet which can make network connection back to the server but is only allowed to read files in the /tmp directory. The execution of this applet may lead to the disclosure of sensitive data to the remote hacker who wrote the two applets.

2.2. Inadvertent execution of a mobile code from an untrusted source

As a third example, we will show that a network user may execute a malicious mobile code containing Trojan Horses and consequently controlled by the Trojan Horses. Limiting accesses to untrusted servers cannot

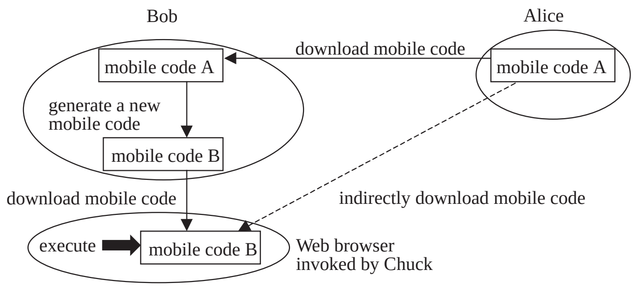


Fig. 4. Execution of indirectly downloaded mobile code.

fully guarantee the security of network users. The Web server may contain a mobile code migrated from other hosts, which contains Trojan Horses. For example in Fig. 4, user Alice creates mobile code A containing a Trojan Horse on a Web server. User Bob may execute mobile code A and unbeknownstly generate a new mobile code, or infect an existing code B. A third user Chuck, who trusts Bob but does not trust Alice, views Bob's Web page and executes mobile code B. Consequently, Chuck is under the control of the mobile code migrated from Alice. In this example, a Web user may be indirectly attacked by a remote untrusted server through a server he trusts. Similar scenarios may appear on proxy servers. The Web proxy server provides efficient caching of other Web servers on the Internet. A proxy server in a firewall protection scheme may also act as a representative for users behind the firewall to access the Web. Users rely on the proxy server to access the Web, but the applets of the original server could contains a Trojan Horse, which has the ability to access the users' system resources indirectly. Thus, an untrusted server can access the browser's resources indirectly.

2.3. Authorization mechanisms for delegated access

Mobile code systems like WWW use a hypertext approach to allow users to navigate the entire information space, where every piece of information is connected via links to related pieces of information [Samarati96]. A user who has access to an object can activate all links to other objects. In Apache [Apache99] and NCSA [NCSA95] HTTP servers, it is possible to restrict access to the information contained in a directory to specific hosts or authenticated users, and consequently the traversal will fail if the user does not have the authorization. This approach does not protect the relationships between objects. For example, suppose that a user is permitted to browse directory A but not directory B. If there exists a file C which is linked to both directories A and B, the user may indirectly access directory B via the link file C. To cope with the authorization problem, Samarati et al. proposed a model [Samarati96] which takes into consideration the relationships between linked objects, and allows administrative privileges to be delegated. It is very difficult in this model to set up a uniform access privilege for users to access each object through different delegates over different link paths, because it is a very complex issue to administrate and unify the access privileges for traversals over different link paths to an object. Inadequate setting of access privileges to an object may leave trap doors. Thus, the effect of a seemingly innocuous access to an object over a path may actually represent an illegal access.

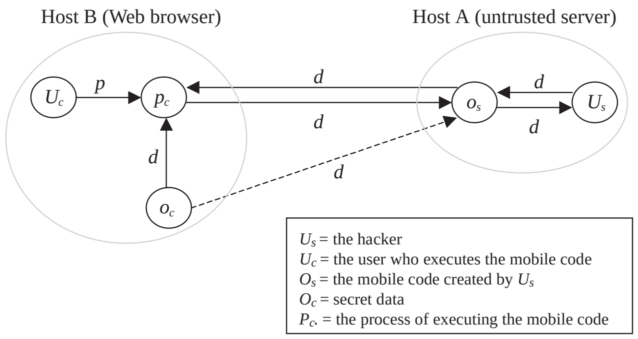
The above examples show that seemingly innocuous access patterns may, in fact, represent illegal accesses whenever they appear in *the context of specific sets of subjects and objects*.

3. Data and privilege flows

In this section, we give the definition of data and privilege flows. The data and privilege flows within a mobile code system can be represented by a directed graph which consists of a set of vertices and a set of labeled edges. The set of vertices consists of subjects and objects. Here, subjects can be users and processes. Objects can represent programs, files, directories, etc. The formal definition of subjects and objects is given in the next section. Each directed edge e_i connects elements of an ordered pair of vertices $\{u, v\}$, and is labeled with subsets of a finite relation set R, where $R = \{d, p\}$. (When written as labels on a graph, the set braces of relations are normally omitted.) The basic accesses can be divided into two types of accesses depending on whether the accesses cause a flow of data or privilege. *Data flows* when read, write, send, and receive operations are invoked. The data flow from a subject/object v_i to another subject/object v_j is denoted by $v_i \xrightarrow{d} v_j$. The *privilege flow* from subject v_i to subject/object v_j, is denoted by $v_i \xrightarrow{p} v_j$, when the former is executing instructions provided by the latter, that is, v_j controls the execution of v_i. Formal data and privilege flow rules are established by analysis of system level operations (that is, system calls). The following interpretation of data and privilege flows between subjects/objects are applicable not only to UNIX but also to other operating systems.

- *read* primitive: (s_i reads o_j) $\Rightarrow o_j \xrightarrow{d} s_i$
- *write* primitive: (s_i writes o_j) $\Rightarrow s_i \xrightarrow{d} o_j$
- *execute* primitive: (s_i is executing o_j) $\Rightarrow s_i \xrightarrow{p} o_j$

Other primitives can also be interpreted in terms of the primitives above. For example, sending a message to a remote process can be interpreted as writing to a remote process. Indirect flow relations are derived from interaction between subjects and objects and their corresponding direct flow relations. The following rules illustrate how indirect flow relations can be derived from the invocation of a new direct flow relation.



Host B (Web browser)
Host A (untrusted server)
U_c
p_c
o_c
o_s
U_s
p
d
U_s = the hacker
U_c = the user who executes the mobile code
O_s = the mobile code created by U_s
O_c = secret data
P_c = the process of executing the mobile code

Fig. 5. Data and privilege flows of the example in Fig. 1.

Rule 1: If $o_i \xrightarrow{d} o_j$ precedes $o_j \xrightarrow{d} o_k$ then $o_i \xrightarrow{d} o_k$ exists.

Note that if $o_i \xrightarrow{d} o_j$ succeeds $o_j \xrightarrow{d} o_k$ then $o_i \xrightarrow{d} o_k$ does not hold.

Rule 2: If $s_i \xrightarrow{p} o_j$ and $s_i \xrightarrow{d} o_k$ appear simultaneously, then $o_j \xrightarrow{d} o_k$ exists.

For example, if user s_i executes a program o_j to writes some data to a file o_k, then there is an implicit data flow from o_j to o_k.

Rule 3: If $o_i \xrightarrow{d} o_j$ precedes $s_k \xrightarrow{p} o_j$ then $s_k \xrightarrow{p} o_i$ exists.

Note that to capture data flow dependencies among subjects and objects, the specific sequencing of flow relations must be included. For example, if process 1 writes a file *before* process 2 reads it, process 1 has a data flow relation with process 2. However, if process 1 writes the file after process 2 reads it, no data flow relation can be established between the two processes.

According to the definition above, we can derive the data and privilege flows of the example in Fig. 1, which is depicted in Fig. 5. When a remote user U_c executes the mobile code, an implicit data flow $o_c \xrightarrow{d} o_s$ is obtained between secret data O_c and mobile code O_s. In the following section, we propose a flow control model to detect all the data and privilege flows, including explicit and implicit. If the detected flows do not violate the rules of access control, then the action caused the flows is allowed, else it is denied.

4. The flow control model

In this section, we provide a succinct definition of the flow control model by formally defining protection states, and state transitions. The protection state for flow control is defined by a five tuple (S, O, Df, Pf, F), where:

- S is a set of subjects, which are the active entities of the model. Subjects can initiate operations and pass data and privileges, and can represent processes and users.

- O is a set of objects, which are the passive entities of the model. Objects can represent memory segments, programs, files, directories, etc. Subjects can also be considered as objects, that is $S \subseteq O$.
- Df is a collection of subject sets. $Df[o]$ represents the set of the subjects that have data flows into object o.
- Pf is a collection of subject sets. $Pf[s]$ represents the set of the subjects that take the privilege of subject s.
- F is a flow control matrix, with rows corresponding to subjects and columns to objects. An entry $F[s,o]$ lists the access privileges of subject s for object o, and $F[s,o] \subseteq \{d_{\text{out}}, d_{\text{in}}, p, Own\}$.

The flow control matrix F defines the flow policy of the mobile code system. An entry $F[s,o]$ lists the legal data and privilege flows between s and o. Note that the flow control policy may *not* be transitive or symmetric, because it is common that the flow policy allows the flows from a to b, and b to c, but not a to c, or b to a. We will use the flow control matrix F to distinguish an illegal access from a legal access, where an access is legal if the incurred flows are permissible by F. The legal-flow policy enforced by F is similar to discretionary and nondiscretionary access control policies. The latter only defines legal direct flow of data and privileges, while the former defines both legal direct and indirect flows of data and privileges. A flow violation occurs when an access incurs flows that violate the given restriction of data and privilege flows; consequently the access should be rejected. For example, an indirect data flow from a remote user to a local system command violates F and thus is illegal.

The purpose of the flow control policy is to control the flows of data and privilege. That is, we need to control the three types of flows: (1) d_{out}—data flows out of an object, (2) d_{in}—data flows into an object, and (3) p—privilege flows out of a subject. Changes to the protection state are modeled by a set of commands which appear in the execution of mobile codes. Commands are specified by a sequence of primitive operations that change the flow control matrix, as well as data and privilege flows between subjects and objects. These operations are conditioned on the presence of certain privileges in the flow control matrix and are controlled by a monitor responsible for managing the protection state. Their effect on the protection state is formally defined in Table 1. Let op be a primitive operator, and $Q = (S, O, Df, Pf, F)$ be the protection state. The execution of op in state Q causes a transition from Q to the state $Q' = (S', O', Df', Pf', F')$ under the conditions defined in Table 1. This is written as $Q|opQ'$ (read "Q derive Q' under op"). Next, we will discuss the commands used in defining the flow control policy.

Any subject may create a new object. The subject creating an object is automatically given ownership of the object, d_{in}-flow and d_{out}-flow rights to the object. This is represented by the command:

```
**command** *create.object*$(s, o)$
    **create object** $o$,
    **enter** $Own$ into $F[s,o]$,
    **enter** $d_{\text{in}}$ into $F[s,o]$,
    **enter** $d_{\text{out}}$ into $F[s,o]$,
```

Any user may create a new subject. The execution of the newly created subject is controlled by the creating user. For example, a user s may create a process s'. The process inherits the user's rights and its execution is controlled by the user. Thus, we have a privilege flow from user s to process s'. This is represented by the command:

```
**command** *create.subject*$(s, s')$
    **create subject** $s'$,
    **enter** $s$ into $Pf[s']$,
    $\forall o \in O$, **if** $Own \in F[s,o]$, **enter** $Own$ into $F[s',o]$,
    $\forall o \in O$, **if** $d_{\text{in}} \in F[s,o]$, **enter** $d_{\text{in}}$ into $F[s',o]$,
    $\forall o \in O$, **if** $d_{\text{out}} \in F[s,o]$, **enter** $d_{\text{out}}$ into $F[s',o]$,
    $\forall o \in O$, **if** $p \in F[s,o]$, **enter** $p$ into $F[s',o]$.
```

The process owning an object can confer any right (except ownership) to other processes. For example, d_{out}-flows may be conferred by process s_1 on process s_2 with the command:

```
**command** *confer.*$d_{\text{out}}(s_1, s_2, o)$
    **if** $Own$ in $F[s_1, o]$
    **then enter** $d_{\text{out}}$ into $F[s_2, o]$
```

(Similar command confers d_{in}-flows.)

A user can also confer the right of a process to an object to control its execution. For example, p-flows of process s' may be conferred by user s to object o with the command:

```
**command** *confer.p*$(s, s', o)$
    *if Own in* $F[s, s']$
    *then* **enter** $p$ into $F[s', o]$
```

The owner of an object may revoke flow rights to the object. Commands for removing access rights from the flow control matrix are similar to those for conferring rights; for example, process s_1 may revoke d_{out}-flow with the command:

```
**command** *revoke.*$d_{\text{out}}(s_1, s_2, o)$
    **if** $Own$ in $F[s_1, o]$
    **then delete** $d_{\text{out}}$ from $F[s_2, o]$
```

(Similar command revokes d_{in} and p-flows.)

Table 1
Primitive operations

Op	Conditions	New state
Enter d_{in} into $F[s,o]$	$s \in S$ $o \in O$	$S' = S$, $O' = O$, $Pf' = Pf$, $Df' = Df$ $F'[s,o] = F[s,o] \cup \{d_{\text{in}}\}$ $F'[s_1,o_1] = F[s_1,o_1]$, $(s_1,o_1) \neq (s,o)$
Enter d_{out} into $F[s,o]$	$s \in S$ $o \in O$	$S' = S$, $O' = O$, $Pf' = Pf$, $Df' = Df$ $F'[s,o] = F[s,o] \cup \{d_{\text{out}}\}$ $F'[s_1,o_1] = F[s_1,o_1]$, $(s_1,o_1) \neq (s,o)$
Enter p into $F[s,o]$	$s \in S$ $o \in O$	$S' = S$, $O' = O$, $Pf' = Pf$, $Df' = Df$ $F'[s,o] = F[s,o] \cup \{p\}$ $F'[s_1,o_1] = F[s_1,o_1]$, $(s_1,o_1) \neq (s,o)$
Enter s' into $Df[o']$	$s' \in S$ $o' \in O$	$S' = S$, $O' = O$, $Pf' = Pf$ $F'[s,o] = F[s,o]$, $s \in S$, $o \in O$ $Df'[o'] = Df'[o'] \cup \{s'\}$ $Df'[o] = Df'[o]$, $o \neq o'$
Enter s_1 into $Pf[s_2]$	$s_1, s_2 \in S$	$S' = S$, $O' = O$, $Pf' = Pf$ $F'[s,o] = F[s,o]$, $s \in S$, $o \in O$ $Pf'[s_2] = Pf[s_2] \cup \{s\}_1$ $Pf'[s] = Pf[s]$, $s \neq s_2$
Create subject s'	$s' \notin S$	$S' = S \cup \{s'\}$, $O' = O \cup \{s'\}$ $F'[s,o] = F[s,o]$, $s \in S$, $o \in O$ $Pf'[s] = Pf[s]$, $s \in S$ $Df'[o] = Df[o]$, $o \in O$ $F'[s',o] = \phi$, $o \in O'$ $F'[s,s'] = \phi$, $s \in S'$ $Pf'[s] = \phi$, $Df'[s] = \phi$
Create object o'	$o' \notin O$	$S' = S$, $O' = O \cup \{o'\}$ $F'[s,o] = F[s,o]$, $s \in S$, $o \in O$ $Pf'[s] = Pf[s]$, $s \in S$ $Df'[o] = Df[o]$, $o \in O$ $F'[s,o'] = \phi$, $s \in S$ $Df'[o'] = \phi$
Destroy subject s'	$s' \in S$	$S' = S - \{s\}$, $O' = O - \{o\}$ $F'[s,o] = F[s,o]$, $s \in S'$, $o \in O'$ $Pf'[s] = Pf[s]$, $s \in S'$ $Df'[o] = Df[o]$, $o \in O'$
Destroy object o'	$o' \in O$ $o' \notin S$	$S' = S$, $O' = O - \{o\}$ $F'[s,o] = F[s,o]$, $s \in S'$, $o \in O'$ $Pf'[s] = Pf[s]$, $s \in S'$ $Df'[o] = Df[o]$, $o \in O'$

The flow rights may be changed if action *read* is invoked which causes a d_{out}-flow of data. For example, the action that subject s reads object o not only causes new data flow into the subject but also changes the flow rights of the subject for other objects. In this example, the flow rights associated with the object can propagate along with the data, and can affect the flow rights of the subject invoking the action. This is represented by the command:

```
command $action.read(s,o)$
  if $s$ reads $o$
  then $\forall s' \in Df[o]$, enter $s'$ into $Df[s]$,
      $\forall s' \in S$,
        if $d_{\text{out}} \in F[s',s]$ but $d_{\text{out}} \notin F[s',o]$
        then delete $d_{\text{out}}$ from $F[s',s]$
end
```

The invocation of action *execute* may cause new privilege flows. For example, the action that subject s executes object o causes new privilege flows into the subjects which have data flows into object o. This is represented by the command:

```
command $action.execute(s, o)$
    if $s$ executes $o$
    then $\forall s' \in Df[o]$, enter $s'$ into $Pf[s]$
    end
```

The invocation of action *write* may cause new privilege flows and change flow rights. For example, the action that subject s writes object o not only causes new data flows into the objects but also changes the flow rights of object o. This is represented by the command:

```
command $action.write(s, o)$
  if $s$ writes $o$
  then $\forall s' \in Df[s]$ or $Pf[s]$, enter $s'$ into $Df[o]$
       $\forall s' \in S$,
           if $d_{\text{out}} \in F[s', o]$ but $d_{\text{out}} \notin F[s', s]$
           then delete $d_{\text{out}}$ from $F[s', o]$
end
```

4.1. Enforcing a flow control policy

A flow control policy, which specifies the legal flows of a system, is expected to cooperate with access control mechanisms to ensure security of mobile code network systems. For example, in Word Wide Web systems, the flow control policy can be enforced in access control unit of Java virtual machine to prevent Web browsers from executing malicious Java applets. An access is permitted if the flows caused by the access are legal. The flow control policy is enforced by validating every user access for appropriate flow rights. Every object has a monitor that validates all accesses to that object in the following manner.

1. A subject s requests an access that causes α_i-flow to object o, where $i = 1, \ldots, n$.
2. The protection system presents triplet (s, α_i, o) to the monitor of o.
3. The monitor looks into the flow rights of s to o. There are three possible cases:
   (1) $\alpha_i = d_{\text{in}}$

   If $d_{\text{in}} \in F[s', o] \forall s' \in Df[s]$, thentheflowispermitted; elseitisdenied.

   (2) $\alpha_i = d_{\text{out}}$

   If $d_{\text{out}} \in F[s, o]$, then the flow is permitted; else it is denied.

   (3) $\alpha_i = p$

   If $p \in F[s, s'] \forall s' \in Df[o]$, then the flow is permitted; else it is denied.

4. If all α_i-flow, where $i = 1, \ldots, n$, are permitted, the access is permitted.

In order to validate the legality of each user access, the flow control policy may cause lots of flow specifications that incur extra system load. For some systems with low computation power, this system overhead may be not affordable. Therefore, the proposed flow control model is recommended for the systems with high security requirement.

4.2. Implementation consideration

Direct implementation of the flow control matrix F as a two-dimensional structure is generally impractical due to the sparseness of the matrix. Instead, F can be implemented as a set of flow control lists (FCLs). Alternatively, it is possible to implement F as a set of rules which define either legal or illegal flows between a given subject (or group of subjects) and a given object (or group of objects), such as a rule that prohibits the privilege flow from any superuser process to remote users, a rule that prohibits the data flow from remote users to a user's sensitive files. In this paper, we will only investigate the flow control lists.

We can specify object o_i which can be known by authorized users in the *Flow Control List* $((U_1, F_1), (U_2, F_2), \ldots)$, where F_i is the set of flow rights that user U_i has to access o_i. There are three kinds of flow rights $d_{\text{out}}, d_{\text{in}}$, and p, that is, $F_i \subseteq \{d_{\text{out}}, d_{\text{in}}, p\}$. If a flow is incurred that is beyond the flow control lists, a security violation occurs. A variant of the flow control list is defined as follows. Let $d_{\text{out}}[o_i], d_{\text{in}}[o_i]$, and $p[s_i]$ represent respectively the set of users who are authorized to have data flow out of o_i, the set of users who are authorized to have data flow into o_i, the set of users who are authorized to take privilege from s_i (that is, the set of user who are authorized to control the execution of s_i). Let $Df[o_i]$ and $Pf[s_i]$ represent respectively the set of the users who had data flows into object o_i, the set of users who take privilege from s_i (that is, the set of users who currently control the execution of s_i).

The action that process p_i of user U_i reads object o_i causes not only new data flows into the process but also the change of flow rights of the process for other objects. If the flows are legal, that is, $U_i \in d_{\text{out}}[o_i]$, the action is permitted. In this context, the FCL associated with the object can propagate along with the data, and can affect the FCLs of other processes and objects that accept the data. The *read* action can be represented by the command:

```
command $action.read(U_i, p_i, o_i)$
    if $p_i$ reads $o_i$
    then
    $Df[p_i] = Df[p_i] \cup Df[o_i]$, and

    $d_{\text{out}}[p_i] = d_{\text{out}}[p_i] \cap d_{\text{out}}[o_i]$
end
```

In the command, $Df[p_i] = Df[p_i] \cup Df[o_i]$ means that foreign data has flowed into the process, and $d_{\text{out}}[p_i] = d_{\text{out}}[p_i] \cap d_{\text{out}}[o_i]$ means that the data flowed into the query should have at least the same protection as the object from which the data was captured.

The flow rights may also be changed if a process p_i of user U_i invokes a *write* action which causes direct and indirect flows of data into an object o_i. If o_i is not a subject and $Df[p_i] \cup Pf[p_i] \subseteq d_{\text{in}}[o_i]$, the flows are legal and the action is permitted. Or if o_i is a subject owned by user U_j and $U_j \in d_{\text{out}}[p_i]$, the flows are legal and the action is permitted. The execution of action *write* may cause both new flow of data, and change of flow rights. For example, the action that process p_i of user U_i writes object o_i not only causes a new data flow into the object but also changes the FCL of object o_i. This is represented by the command:

```
command $action.write(U_i, p_i, o_i)$
    if $p_i$ writes $o_i$
    then $Df[o_i] = Df[o_i] \cup Df[p_i] \cup Pf[p_i]$, and

    $d_{\text{out}}[o_i] = d_{\text{out}}[p_i] \cap d_{\text{out}}[o_i]$
end
```

In the command, $Df[o_i] = Df[o_i] \cup Df[p_i] \cup Pf[p_i]$ means that foreign data has flowed into the object, and $d_{\text{out}}[o_i] = d_{\text{out}}[p_i] \cap d_{\text{out}}[o_i]$ means that the data flowed into the object should have at least the same secrecy level as the object from which the data was captured. The flow rights will not be changed if a process invokes an *execute* action which causes direct and indirect flows of privilege into an object. If the flows are legal, that is, $Df[o_i] \subseteq p[s_i]$, the action is permitted. This is represented by the command:

```
command $action.execute(U_i, p_i, o_i)$
    if $p_i$ executes $o_i$
    then $Pf[p_i] = Df[o_i] \cup \{U_i\}$
end
```

In the command, $Pf[p_i] = Df[o_i] \cup \{U_i\}$ states that the execution of the process p_i is controlled by its owner U_i and the users who wrote the code.

5. Applications of the model

The access patterns that need to be analyzed for flow control purposes should be characterized in terms of direct accesses and sequences of accesses between subjects and objects. We can use the flow control list to discriminate between an illegal access and a legal access. An illegal access occurs when the access violates the given restriction of data and privilege flows, defined by the flow control list. In the appendix, we will show applications of the model to prevent several types of illegal accesses: the inadvertent modification and disclosure of sensitive data, as well as the inadvertent execution of mobile codes.

Virus can spread rapidly through networks. Once getting into a host, it duplicates itself, seeks files that it can infect, and spreads to other hosts through computer networks. The model can be used to determine whether a host has not been infected by a virus which comes from remote host through networks or not. Herein, we say that a host is *infection-free* if there is no virus that spreads from remote hosts to the host through networks. It is, however, difficult to identify viruses. We choose to solve the problem in a strict sense, that is, if there is no data flow from remote hosts to the object of a host, the site is *infection-free*. To be *infection-free*, a process cannot copy (read and then write), or write while executing a remote mobile code. In the case that a host is infected, it is possible, with the data and privilege model, to identify the origin of the virus.

6. Conclusions

Conventional access control mechanisms are not effective against context-dependent illegal accesses in mobile code systems. In this paper we investigate the context-dependent attacks and present a flow control model which tracks data and privilege flows, and can uniformly define various types of illegal access patterns. Direct implementation of the flow control matrix as a two-dimensional structure is generally impractical due to the sparseness of the matrix. To realize the flow control mechanism, we adopt the same ideas as those used in constructing access control mechanisms, and propose an efficient construction of the flow control lists. The proposed flow control mechanism can be integrated with access control mechanisms to provide better security.

Appendix A

Here, We describe how the proposed model can prevent several types of illegal accesses: the inadvertent modification and disclosure of sensitive data, as well as the inadvertent execution of mobile codes.

A.1. Inadvertent modification of sensitive data

Consider the example given in Fig. 2. A process p_c of an unsuspecting local user U_c inadvertently downloads and executes a mobile code o_s written by a remote user U_s. Both the mobile code and the remote user can acquire the invoker's privilege, thereby controlling all its activities and allowing the remote user to indirectly write over the unsuspecting user's sensitive objects o_c. Consequently, the remote user may modify the unsuspecting user's sensitive object through the mobile code. Table 2 gives the reasons why an action is accepted or rejected, as well as the new flows caused the action if it is accepted. In the example, to prevent the indirect damage by the remote user, the flow control list must be set such that $U_s \notin d_{\text{in}}[o_c]$ (that is, $d_{\text{in}} \notin F[U_s, o_c]$). In this way, the action *action.write*(p_s, o_c) that causes the illegal flow—U_s into $Df[o_c]$—can be rejected.

A.2. Inadvertent disclosure of sensitive data

Consider the example in Fig. 1. A remote user U_s may deliberately leave a mobile code o_s on a Web server to attract the attention of network users. Subsequently, an unsuspecting user U_c downloads and executes the mobile code. While executing the code, the remote user who wrote the code can acquire all the invoker's privilege to read the unsuspecting user's sensitive object o_c and send it back to himself. Consequently, the remote user may acquire the unsuspecting user's sensitive file through the mobile code. In this context, these seemingly normal actions of the mobile code can be prevented by the flow control mechanism. Table 3 gives the reasons why an action is accepted or rejected, as well as the new flows caused the action if it is accepted. In the example, to prevent the indirect disclosure of sensitive object o_c to the remote user, the flow control list must be set such that $U_s \notin d_{\text{out}}[o_c]$ (that is, $d_{\text{out}} \notin F[U_s, o_c]$). Although initially the process p_c executing mobile code o_s is allowed to have outward flow, that is, $d_{\text{out}}[p_c] = U$. However, as a result of the sequence of actions, $d_{\text{out}}[p_c]$ is affected by $d_{\text{out}}[o_c]$. When p_c accepts the data from o_c, $d_{\text{out}}[p_c] = d_{\text{out}}[p_c] \cap d_{\text{out}}[o_c] = \{U_c\}$. In this way, the action *action.write*(p_c, p_s) that causes the illegal direct flow from p_c to U_s, and the illegal indirect flow from o_c to p_s can be rejected.

A.3. Inadvertent execution of a mobile code from an untrusted source

Now consider the example in Fig. 4, user Alice (U_a) creates mobile code A (o_a) containing a Trojan Horse on a Web server. User Bob (U_b) may download and execute it which furtively generates a new mobile code B (o_b). Eventually, a third user Chuck (U_c) views Bob's Web page and executes mobile code B, then Chuck is under the control of mobile code migrated from Alice. The mobile code B may disclose Chuck's secret information (o_c) to Alice or modify his sensitive data. In the example, the flow control mechanism can be applied to resist this kind of intrusions. The reasons given in Table 4 are similar to the two types of illegal accesses mentioned above.

Table 2
Preventing indirect damage to sensitive objects

Sequence of actions	Result	Reason	Cause new flows
Initially,			$Df[o_s] = \{U_s\}$ $Df[o_c] = \{U_c\}$, $Pf[p_c] = \{U_c\}$
action.execute(p_c, o_s),	Accepted	$U_s \in p[p_c]$	$Pf[p_c] = \{U_c, U_s\}$
action.write(p_c, o_c),	Rejected	$U_s \notin d_{\text{in}}[o_c]$	

Table 4
Preventing inadvertent execution of a mobile code

Sequence of actions	Result	Reason	Cause new flows
Initially,			$Df[o_a] = \{U_a\}$ $Df[o_b] = \{U_b\}$ $Pf[p_b] = \{U_b\}$, $d_{\text{in}}[o_b] = \{U_a, U_b\}$ $p[p_c] = \{U_b, U_c\}$
action.execute(p_b, o_a),	Accepted	$U_a \in p[p_b]$	$Pf[p_b] = \{U_a, U_b\}$
action.write(p_b, o_b),	Accepted	$U_a \in d_{\text{in}}[o_b]$	$Df[o_b] = \{U_a, U_b\}$
action.execute(p_c, o_b),	Rejected	$U_a \notin p[p_c]$	

Table 3
Preventing indirect disclosure of sensitive information

Sequence of actions	Result	Reason	Cause new flows
Initially,			$Df[o_s] = \{U_s\}$, $d_{\text{out}}[p_c] = U$ $Df[o_c] = \{U_c\}$, $d_{\text{out}}[o_c] = \{U_c\}$ $Pf[p_c] = \{U_c\}$
action.execute(p_c, o_s),	Accepted	$U_s \in p[p_c]$	$Pf[p_c] = \{U_c, U_s\}$
action.read(p_c, o_c),	Accepted	$U_c \in d_{\text{out}}[o_c]$	$Df[p_c] = \{U_c\}$ $d_{\text{out}}[p_c] = \{U_c\}$
action.write(p_c, o_c),	Rejected	$U_s \notin d_{\text{out}}[p_c]$	

References

[Apache99] Apache Development Group, Apache modules, http://www.apache.org/docs/mod/index.html, February 1999.

[Bern94] T. Berners-Lee, R. Calilliau, A. Luotonen, H.F. Nielsen, A. Secret, The world wide web, Comm. ACM 37 (8) (August 1994) 77–82.

[Cert96a] CERT Coordination Center, Java implementations can allow connections to an arbitrary host, CERT Advisory CA-96.05, ftp://info.cert.org/pub/cert_advi-n.5pt}-
advisories/CA − 96.05.*java_applet_security_mgr, March*1996.

[Cert96b] CERT Coordination Center, Weaknesses in Java bytecode verifier, CERT Advisory CA-96.07, ftp://info.cert.org/pub/cert_advisories/CA-96.07.java_bytecode_verifier, March 1996.

[Dean96] D. Dean, E.W. Felten, D.S. Wallach, Java security: from HotJava to Netscape and beyond, Proceedings of IEEE Symposium on Research in Security and Privacy, Oakland, CA, May 1996.

[Denning76] D.E. Denning, A lattice model of secure information flow, Comm. ACM 19 (5) (May 1976) 236–243.

[Denning77] D.E. Denning, P.J. Denning, Certification of programs for secure information flow, Comm. ACM 20 (7) (July 1977) 504–512.

[Flanagan96] D. Flanagan, Java in a nutshell, O'Reilly & Associates, Inc., February 1996.

[Gosling95] J. Gosling, H. McGilton, The Java language overview: a white paper, Sun Microsystems Technical Report, May 1995.

[Gosling96] J. Gosling, H. McGilton, The Java language environment, Sun Microsystems, http://java.sun.com/doc/language_environment/, May 1996.

[Hot96] HotJava(tm): the security story, Sun Microsystems, http://java.sun.com/1.0alpha3/doc/security/security.html, 1996.

[Lampson73] B.W. Lampson, A note on the confinement problem, Comm. ACM 10 (16) (October 1973) 613–615.

[Landwehr81] C.E. Landwehr, Formal models for computer security, ACM Comput. Surveys 13 (3) (September 1981) 247–275.

[Lipner75] S. Lipner, A comment on the confinement problem, ACM Oper. System Rev. 9 (5) (November 1975) 192–196.

[NCSA95] NCSA httpd Development Team, NCSA httpd, http://hoohoo.ncsa.uiuc.edu/docs/Overview.html, July 1995.

[Saltzer75] J.H. Saltzer, M.D. Schroeder, The protection of information in computer systems, Proc. IEEE 63 (9) (September 1975) 1278–1308.

[Samarati96] P. Samarati, E. Bertino, S. Jajodia, An authorization model for a distributed hypertext system, IEEE Trans. Knowledge Data Eng. 8 (4) (August 1996) 555–562.

[Shieh96] Shiuh-Pyng Shieh, Wen-Her Yang, An authentication and key distribution system for open network systems, ACM Oper. Systems Rev. 30 (2) (April 1996) 32–42.

[Shieh97a] Shiuh-Pyng Shieh, Wen-Her Yang, An authentication protocol without trusted party, IEEE Comm. Lett. 1 (3) (May 1997) 1–3.

[Shieh97b] Shiuh-Pyng Shieh, Virgil D. Gligor, On a pattern-oriented intrusion detection model, IEEE Trans. Knowledge Data Eng. 9 (4) (August 1997) 661–668.

[Shieh99] S.P. Shieh, C.T. Lin, W.B. Yang, and H.M. Sun, Digital multisignature schemes for authenticating delegates in mobile code systems, IEEE Trans. Vehicular Tech. 49 (4) (2000) 1464–1473.

[Yellin95] F. Yellin, Low level security in Java, Fourth International World Wide Web Conference, Boston, MA, World Wide Web Consortium, http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html, December 1995.