Evaluating the Security Of Three Java-Based Mobile Agent Systems

Sebastian Fischmeister, Giovanni Vigna, Richard A. Kemmerer

Department of Computer Science
University of California Santa Barbara
{sfischme,vigna,kemm}@cs.ucsb.edu

Abstract. The goal of mobile agent systems is to provide a distributed computing infrastructure supporting applications whose components can move between different execution environments. The design and implementation of mechanisms to relocate computations requires a careful assessment of security issues. If these issues are not addressed properly, mobile agent technology cannot be used to implement real-world applications. This paper describes the initial steps of a research effort to design and implement security middleware for mobile code systems in general and mobile agent systems in particular. This initial phase focused on understanding and evaluating the security mechanisms of existing mobile agent systems. The evaluation was performed by deploying several mobile agents systems in a testbed network, implementing attacks on the systems, and evaluating the results. The long term goal for this research is to develop guidelines for the security analysis of mobile agent systems and to determine if existing systems provide the security abstractions and mechanisms needed to develop real-world applications.

Keywords: Mobile agent systems, computer security, security testing.

1 Introduction

Recently mobile code has attracted a great deal of interest from both industry and academia. The ability to dynamically deploy application components across the network is a powerful mechanism to improve the flexibility and customizability of applications.

Mobile code is a general concept that encompasses a number of different approaches to reconfigure the location of the components of a distributed application [7]. The most common form of code mobility is *code on demand*, which is the download of executable content in a client environment as the result of a client request to a server. A well-known example of this approach is the download of Java applets or Javascript code in a WWW browser. A different form of code mobility is represented by the upload of code to a server. The uploaded code is executed by the server and possibly the results of the computation are sent back to the client. This form of mobility, also known as *remote evaluation* [13], allows the client to execute a computation close to the resources located at the server's side so that network interaction can be reduced. Common examples are represented by the use of SQL to perform queries on a remote database or the upload of PostScript code to a remote printer. A third form of code mobility is represented by the *mobile agent* paradigm. In this case, mobile components can explicitly relocate themselves across the network, usually preserving their execution state (or part thereof) across migrations. Examples of systems supporting this type of mobility are Telescript [17] and D'Agents [8].

Past research on mobile code security has mainly focused on code on demand and remote evaluation [6]. These forms of mobility are easier to deal with because they encompass a single interaction in the transfer of a code component. Some of the results achieved in these areas have been applied to the mobile agent approach, but the problem of creating a distributed computing infrastructure where agent-based applications belonging to different (usually untrusted) users can execute concurrently has not been solved yet [15]. In most cases, mobile agent systems (MASs) are proof-of-concept prototypes whose focus is on sophisticated mobility mechanisms; security is left as future work. Other systems provide some basic security mechanisms and primitive support for the definition of security policies, but the provided mechanisms are far from being a sound, comprehensive security solution. If the security problem is not solved in a reliable way, the applicability of mobile agent technology in the real world will be impossible.

This paper describes the first steps of a research effort aimed at the development of secure mobile agent systems. As a preliminary phase in this research effort, it was decided to assess the security provided by existing MASs. In this phase a number of MASs that provide security mechanisms were installed on a testbed network in the Reliable Software Lab at UCSB. The network is composed of hosts running various operating systems, such as Sun Solaris 2.x running Sun's reference implementation of the Java Development Kit (JDK) 1.1.8, Linux 2.x running the JDK 1.1.7, and Microsoft's Windows NT 4.0 with JDK 1.1.7. Attacks were launched against the MASs under exam, and the results were analyzed.

The results of the security analysis for a subset of the MASs that were collected and installed are presented in this paper. The subset includes Aglets SDK 1.1, Jumping Beans 1.1, and Grasshopper 1.2.2.3. The remainder of this paper is organized as follows. Section 2 presents some terminology by describing an abstract mobile agent system, and it also reviews some basic security terminology. Sections 3, 4, and 5 present the results of instantiating attacks against the authorization mechanisms of the systems under analysis. Section 6 draws some conclusions and outlines future work.

2 General Framework for the Analysis

Before discussing each of the mobile agent systems it is important to define some common concepts, abstractions, and terminology. This framework will then be used to define some general attack classes, which can then be instantiated on particular systems. The definitions presented in this section were obtained by the analysis of a number of existing systems and their security models [8, 9, 12, 14], as well as by the OMG's MASIF specification [10].

An abstract mobile agent system is shown in Figure 1. The main components are mobile agents, places, agent systems, regions, and principals. A *mobile agent* is a com-

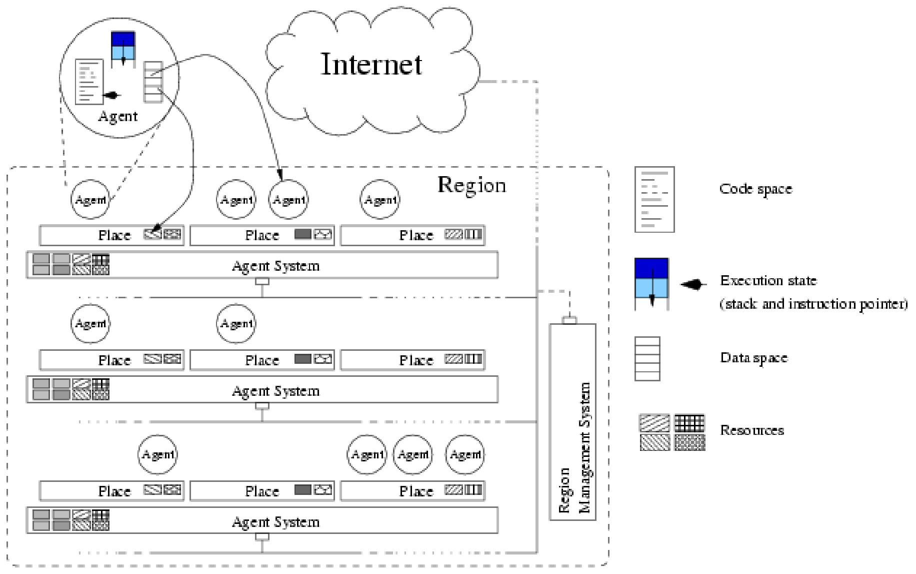

Fig. 1. A mobile agent system.

putational unit and it consists of a *code space,* an *execution state,* and a *data space.* The code space contains a set of references to code fragments that can be invoked during the execution of the agent. The code space includes both references to fragments that are owned by the agent (e.g., the code that specifies the behavior of the application component implemented by the agent) and references to external classes that can be part of a place, system, or region (e.g., the code of procedures that implement system services). The execution state contains all the information related to the evolution of the agent, e.g., the execution stack, the code fragment currently being executed, and the program counter. The data space contains references to external resources that can be accessed by the agent, e.g., a reference to an open file.

The execution of agents is supported by *places.* Each place provides a local infrastructure to a visiting mobile agent. The place infrastructure supports the execution of particular procedures as defined in the associated *code repository* and provides access to local *resources* (e.g., a database or a local printer). Access to local resources is regulated by the place's *security system.* The security system comprises three subsystems whose tasks are authentication, authorization, and accounting. Each subsystem contains a *policy* that specifies how the security functionality is configured (e.g., the CPU quotas to be assigned to incoming agents in the case of the accounting subsystems), a set of *security resources* that represent dynamic information about the state of the system, (e.g., the current credentials for a visiting agent in the case of the authentication system), and a *code repository* that contains the definition of the procedures used to implement the security subsystem mechanisms.

Several places may be grouped within an *agent system.* Places inside an agent system may share resources, code, or security mechanisms and, in general, have a privileged relationship with each other. Moving an agent between places in the same agent system and interaction among agents within the same agent system is considered less expensive than interaction or mobility between different agent systems. Usually an agent system is implemented on a single host. An agent system has a structure that is similar to the structure of a place. Its resources, code repository, and security system are shared by the contained places. For example, the authentication system of a place may define its authentication procedures on the basis of those defined at the agent system level.

Agent systems may be grouped in *regions.* A region represents a security domain where network-wide resources are accessed following a uniform policy. Like places, and agent systems, a region is defined in terms of code repository, resources, and security systems. For example the accessible nodes within the region may be specified as resources at the region level. As another example, role-based access control policies may be specified at the region level and then enforced locally by the agent systems.

Agents, places, systems, and regions are associated with a number of *principals* that represent real-world entities such as a person, an organization, or a company. Principals are responsible for the definition or the actions of a specific component of a region (e.g., see [9]). Principals may be associated with particular tasks or responsibilities and their definition may span a place, a system, or a region. For example, a principal may be responsible for the definition of the code fragments used to check the identity of a moving agent inside a region, or it may identify the owner of a resource available at a place.

Traditionally security mechanisms have been classified into authentication mechanisms, authorization mechanisms, and accounting (or resource control) mechanisms. Authentication mechanisms determine who the principal(s) associated with a particular component in a system is (are). Authorization mechanisms determine the acceptable actions of a component on the basis of its associated principal, as determined by the authentication process. The set of possible actions is specified by a policy that given a subject, an object, and the action to be performed, specifies if the requested access should be granted or not. Accounting mechanisms regulate the amount of resources that can be accessed by a component and may be used as a basis for billing procedures. In this paper the analysis is limited to authorization mechanisms.

Authorization mechanisms are analyzed by means of an *access matrix.* Intuitively, the access matrix helps to determine what the possible *access space* for a component is: that is, what other components in the model can be accessed, e.g., by means of an object reference or a file descriptor. The access matrix contains rows and columns labeled with the components of the model. Each cell in the matrix holds the type of access that the component referenced in the corresponding row is allowed for the component referenced in the column. The type of access can be *direct, indirect,* or *non existent.* Direct access implies that access can be performed through a direct reference, e.g., through an object reference. An indirect reference specifies that access to the object is implicit by means of a system/subsystem relation or some other association. For example, the execution state of an agent may be indirectly accessible by the agent itself, even though the agent has no means to access a representation of its stack directly. This could be accomplished, for example, by having the agent access it indirectly by means of diagnostic and exception handling routines.

An access matrix can be *local*, *remote*, or *external*. A local access matrix describes access to elements in the same place or system. A remote access matrix specifies access to components in different systems of the same region. An external matrix describes the case where access can cross protection domain boundaries.

The analysis of a system is performed by analyzing the different access matrices and filling in the types of access allowed between components implemented in the particular system. For each possible access, one must determine what are the possible operations and what subset of these operations would actually be permitted. Then each operation is exercised and the outcome is verified against the defined policy.

In the following, we present the results of analyzing the authorization security for three Java-Based systems. All three of these systems use access control lists (ACLs) to implement the access matrix.

3 Aglets

The Aglets Software Development Kit [2] (Aglets SDK) is a Java-based mobile agent system developed by IBM Tokyo Research Laboratory in Japan. The version analyzed and evaluated in this paper is the beta version, 1.1 beta2. Recently, Aglets became an open source project. Its current release is 2.0b.

3.1 The Aglets Model

In the Aglets SDK mobile agents are called "aglets". The code space of an aglet contains a set of private Java classes (the implementation of the aglet) and references to classes in the runtime system. Aglets are implemented as threads in a Java Virtual Machine and their execution state is represented by the thread's stack and the corresponding program counter. The data space of an aglet contains references to system resources (e.g., sockets and files) and references to other aglets or to local objects that act as wrappers to provide access to particular resources (e.g., a database). Although the Aglets model does distinguish between places and agent systems, the software that is shipped with the system does not support multiple contexts. A single place inside a single agent system is mapped to a component called the *"Tahiti"* server. Regions are not present in the Aglets' systems. The mapping from our abstract model to Aglets is shown in Table 1.

The Tahiti server supports agent execution, provides mechanisms for agent mobility, and implements the security mechanisms. The code repository for the Tahiti component is a set of Java classes that implement the runtime system. Local resources are implemented as stationary agents or object wrappers. The Aglets agent system provides a simple authentication subsystem based on host identifiers and no accounting or resource control system is provided. Authorization is enforced by an implementation of the Java Security Manager interface. The Aglets system defines a policy description language to define access control lists for resources such as files, sockets, and runtime objects. These ACLs can be configured depending on the agent's source host. For details about available permissions see the Aglets white paper [11].

Model	**Aglets**
Mobile Agent	Aglet
Place	Context
Place Resources	Internal objects and Aglets
Agent System	Tahiti
Agent System Resources	Internal objects and Aglets
Region	missing

Table 1. Realization of the Abstract Model in Aglets

3.2 Authorization Attacks in Aglets

Some attacks have already been identified by developers [11] or have been theoretically shown in research papers [16]. So this paper only describes attacks that were novel at the time of the tests.

Code repository attacks. Starting with the access control list, an attack to obtain a reference to the code repository of the Aglets system was attempted. The code repository is not directly accessible by the agent through a reference, and, therefore, it was necessary to obtain the associated information indirectly. We found that by using the Java reflection classes it was possible to disclose information about the system's code repository. To perform the attack, the agent first throws an exception. The exception stores a snapshot of the current execution stack trace. The stack trace stored in the exception is then analyzed and all the class names referenced in the stack are stored for further processing. Once a number of classes have been identified, the Java reflection classes are used to obtain the constructor, attributes, methods, interfaces, and superclass of the class. By examining the signatures of the methods, more classes are found. These classes are added to the ones found in the first phase. The discovery process stops after each class has been analyzed and stored. At this point portions of the code have been revealed. In the final phase the classes are examined to find if there are any static methods or attributes. These are particularly useful because they allow an agent to perform operations without the need for an object reference.

Security policy attacks. The access gained with the previous attack established the basis for an attack against the security policy component. More precisely, we found that the policy database can be accessed by using a static method. This means that it is possible to access the policy database even without having any reference to the policy object. This is not a problem per se, but when write access to the policy object was attempted, it was found that modifications to the policy database are not checked by the Security Manager. So it is possible to add or modify all policies without getting any security exceptions, effectively compromising the security of the system.

Graphic user interface attacks. Part of the analysis focused on the possibility of an agent accessing the graphic system of the Aglets platform. In fact, access to the graphic interface allows an agent to interact with the user sitting at the host graphic console.

In principle, the Aglets system only allows agents to create windows with a warning banner. This is to prevent a malicious agent from spoofing legitimate applications (e.g., a login prompt) that may be used to induce the user to insert sensitive information. We found that, due to a bug in the implementation, the permission *“showWindowWithoutWarningBanner”* is completely useless. Although an agent is not granted the permission, the agent is able to open frames and dialogs (neither of them includes a warning label). This vulnerability was exploited by creating a spoofed login prompt that simulated an operating system request for user authentication. The agent would then obtain username and password and mail them back to the user.

4 Jumping Beans

Jumping Beans [4], developed by AdAstra Engineering, is a commercial framework for implementing mobile agent applications. The analysis in this paper is based on version 1.1. The current version is 2.1.1.

4.1 The Jumping Beans Model

In the Jumping Beans framework mobile agents are called “Mobile Applications”. The code space of a mobile agent includes application-specific Java classes and classes that are part of the Java runtime system. A mobile agent component is implemented as a Java thread and the associated execution state is the thread stack and program counter. The data space may include references to other agents or to external objects.

Jumping Beans does not distinguish between agent systems and places. An agent system instance is called “Agency” and provides only one place. The code repository for an agency includes Java classes for the runtime and site-specific classes for the implementation of local services. Agency resources can be implemented in two ways: they are either represented by mobile agents or they are directly bound to the agent system. It is possible to define one agent system local object per instance. For example, the object may be used to implement a broker service or a wrapper for an external database.

Jumping Beans provides the concept of region. A region is controlled by a component called “Server”. Agent systems within a region have to register with the server, which maintains access control lists for region resources and monitors agent systems and agents in a centralized way. Table 2 provides an overview of the mapping between our abstract model and Jumping Beans.

4.2 Authorization Attacks in Jumping Beans

Jumping Beans implements an authorization system that supports access control lists for certain resources (e.g. network, file system, etc.). For a more detailed list see the Jumping Beans white paper [5]. The agent system policies are set by the administrator through the region’s server. Authorization is enforced by the agent systems. The agent system receives the ACLs from the region server and enforces them through an implementation of the Java Security Manager. Starting from version 1.1 Jumping Beans also

Model	**Jumping Beans**
Agent	Mobile Application
Place	Agency
Place Resources	Internal objects and Agents
Agent System	Agency
Agent System Resources	Internal objects and Agents
Region	Server

Table 2. Realization of the Abstract Model in Jumping Beans

includes a role model for agent system owners. Therefore, it is possible to define access control lists for groups and to assign users to groups. Every mobile agent has a separate permission set and access control list. As a consequence of mobile agent migration, an agent's permissions may get more restrictive, but never less restrictive.

Unauthorized access to the contents of code fragments is implemented by bytecode obfuscation and "final" classes, which are classes that cannot be subclassed. Both mechanisms are not reliable. Bytecode obfuscation makes it harder to reverse engineer Java bytecode, but does not prevent it; a determined attacker may successfully decompile and reverse-engineer the Java classes. The final class mechanism was successfully attacked by removing the final flag in the obfuscated bytecode and creating a malicious subclass. Although the bytecode is obfuscated it is still possible to disclose data, code, and flow control by using the exception mechanisms and the reflection functionalities provided by the Java runtime, as discussed in the previous section for Aglets.

In addition, as mentioned before, Jumping Beans uses the least trust principle (an agent can only become more restricted). However, when analyzing the implementation of the least trust principle, we discovered that it had been implemented without any exceptions. Because of this, the mechanism can be exploited to perform an attack against the access capabilities of a server. To be more specific, if an agent removes all access privileges to itself, then it is impossible even for the region controller to remove the agent from the target agent system. The agent system's state has to be manually deleted (otherwise, the agent would be restarted after a reboot) and the system has to be restarted.

Graphic user interface attacks. In analyzing the access to the graphic system we found that the GUI is implemented in a separate thread. Because of this, after a window has been opened it no longer belongs to the agent. So it is possible for an agent to open window frames and move onto the next host. After migration, all the windows that were opened remain open. The successful implementation of this attack opens a window the size of the whole screen. This window cannot be closed except by closing the entire virtual machine, disabling the agent system.

Runtime system calls attacks. In attempting to access the system's runtime code repository a complete check of the available system-related calls was performed. The Security Manager blocked most of the attempts but, due to an incomplete implementation of the Security Manager, it was possible to invoke the static method *"System.exit()"*, which is the exit routine provided by the Java runtime. The net effect of this call is to shut down the whole system[1].

5 Grasshopper

The Grasshopper mobile agent system [1] is developed by GMD FOKUS and distributed by IKV++ [3]. Grasshopper is the reference implementation for the OMG's MASIF specification [10]. The current version of Grasshopper is 2.2. The analysis of the system was performed using Grasshopper version 1.2.2.3.

5.1 The Grasshopper Model

The Grasshopper model closely follows the one described in the MASIF specification. A mobile agent is called *"Service"* and an agent system is called *"Agency"*. Agencies contain *"Places"* and are organized in *"Regions"*. The basic infrastructure is accessible via the agent system and the local infrastructure has to be implemented in separate agents. The mapping from our abstract model to Grasshopper is shown in Table 3.

Model	**Grasshopper**
Agent	Service
Place	Place
Place Resources	Agents
Agent System	Agency
Agent System Resources	Internal objects and Agents
Region	Region

Table 3. Realization of the Abstract Model in Grasshopper

5.2 Authorization Attacks in Grasshopper

Grasshopper's authorization system is similar to the one implemented in the Aglets SDK. However, the implementation of the Java Security Manager in Grasshopper is incomplete.

Trusted code base attacks. Similar to the Aglets SDK, Grasshopper uses trusted classes. These classes override the Security Manager and are not checked for access. In the case of Grasshopper this leads to a security leak. The third party trusted class *javax.swing.JInternalFrame* can be used to exit the virtual machine. Therefore, it is possible to exit the server.

[1] This attack also bypasses the persistency mechanisms built into the system, making recovery impossible.

Graphic user interface attacks. In analyzing the access to the graphic system we found that the *checkAwtEventQueueAccess* method has not been implemented. By exploiting this vulnerability it was possible to access the event queue associated with the graphic interface and trace the graphic events. Through the event references it was possible to obtain a handle to graphic components external to the agent. The components were then controlled by sending spoofed events. An attack that sends the key-code “Alt-Shift-Q”, which quits the Grasshopper agent system, was implemented. The attack also monitors the event queue for the appearance of a dialog asking the user to acknowledge the quit command and sends a return key event, simulating the “confirmation click”. By doing this, it was possible to bypass the authorization system and to shutdown the Grasshopper agent system.

System properties attacks. By analyzing access to system properties we found that there is no security check on calls to the *checkPropertyAccess* method. By exploiting this vulnerability it is possible to access and modify any property that is available in the system, for example `agency.name`, `agentsystem.protocol`, `region.registry.host`, or `user.home`.

Policy system attacks. When trying to test the access to the policy system we found that, similar to the Aglets system, the policy is accessed through static methods and variables. Although access to the policy object is successfully enforced and special permissions are needed to access the policy object, it is still possible to instantiate a new policy object. Since the policy object is static, the new instance is automatically the valid policy. Although it does not immediately affect the system, the new policy will affect the system the first time that the system manager opens the policy configuration dialog.

6 Conclusions

This paper presented some initial results of a research effort aimed at the analysis of the security issues in mobile agent systems. Three Java-Based mobile agent systems implementing security mechanisms were installed on a testbed network, these systems were analyzed, and numerous attacks were launched against them. The analysis found many interesting vulnerabilities.

The long term goal of this study is to understand the security issues in MASs and to provide a reference model that can help in abstracting security mechanisms and in defining attack classes in a way that is independent of a particular technology. By doing this, the security analysis results can be reused as guidelines to evaluate the security of other MASs. In addition, the use of a reference model highlights the security *abstractions* available in the different languages. Complex applications may require sophisticated security abstractions such as policies, different types of principals, and so on. If these concepts are not available, they have to be developed on top of the existing system, which is usually time-consuming and error-prone.

In this paper we concentrated on the attacks performed by a mobile agent against the authorization mechanisms. Many other attacks were suggested by the analysis, and other systems have been installed in the testbed network. Future work will focus on completing the security analysis of the additional systems and in developing a reference model.

The next step in this research effort will be to build on the experience gained from the security analysis and develop guidelines for the design and development of secure mobile agent systems. Eventually the guidelines will be used to develop a secure agent system that could be effectively used to develop mission-critical mobile agent applications.

References

1. Grasshopper. WWW Site. `http://www.grasshopper.de`.
2. IBM Aglet Workbench. WWW Site. `http://www.trl.ibm.co.jp/aglets/`.
3. IKV++. WWW Site. `http://www.ikv.de`.
4. Jumping Beans. WWW Site. `http://www.jumpingbeans.com`.
5. AdAstra. Jumping Beans White Paper. Technical report, AdAstra Engineering, Inc., Sunnyvale, CA, April 27 1999.
6. D. Dean, E. Felten, and D. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proc. of the 1996 IEEE Symp. on Security and Privacy*, Oakland, Cal., May 1996.
7. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.
8. R.S. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in Multiple-Language, Mobile-Agent System. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer, 1998.
9. G. Karjoth, D. Lange, and M. Oshima. A Security Model For Aglets. *IEEE Int. Comp.*, pages 68–77, July 1997.
10. OMG. MASIF - Mobile Agent System Interoperability Facility. Draft, October 3 1998.
11. M. Oshima, G. Karjoth, and K. Ono. Aglets Specification 1.1 Draft. Whitepaper Draft 0.65, Sept. 8 1998.
12. J. Ousterhout, J. Levy, and B. Welch. The Safe-Tcl Security Model. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer, 1998.
13. James W. Stamos and David K. Gifford. Implementing Remote Evaluation. *IEEE Trans. on Soft. Eng.*, 16(7):710–722, July 1990.
14. G. Vigna. *Mobile Code Technologies, Paradigms, and Applications*. PhD thesis, Politecnico di Milano, 1997.
15. G. Vigna, editor. *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer, 1998.
16. J. Vitek, M. Serrano, and D. Thanos. Security and Communications in Mobile Object Systems. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, LNCS 1222. Springer-Verlag, April 1997.
17. J.E. White. Telescript Technology: Mobile Agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1996.