Survey of Security in Mobile Code Systems

Fritz Schneider and Mayoor Savla
Dept. of Computer Science and Engineering
University of California, San Diego
(fritz@cs.ucsd.edu, msavla@gateway.net)

Abstract
Mobile software agents represent a relatively new paradigm in distributed computing. This new technology poses security problems over and above those traditionally associated with distributed systems. In this paper we present these concerns and conduct a general survey of solutions proffered by the current literature.

1 Introduction

Mobile code is a relatively new paradigm in the arena of distributed software systems. It extends the traditional notions of message passing and remote procedure calls by enabling heterogeneous and potentially untrusted hosts to exchange executable code. Java applets will be familiar to the reader as a very basic illustration of the concept.

We define *mobile code* as executable code that can move from one computer to another. *Strong mobility* refers to the ability to migrate an entire executing process or thread (including its associated data and state) from one networked computer to another. *Weak mobility* refers to the transportation of executable code only (e.g., Java applets or ActiveX controls)[4]. A *mobile agent* is a process that can achieve autonomous strong mobility.

Mobile agents pose serious security problems for both clients (creators of agents) and servers (hosts upon which agents execute). Client agents execute at the mercy of remote servers. They may be subject to modification or mistreatment at the hands of an errant or malicious host. Servers face a similar problem: they are asked to execute code of questionable or even unknown origin, quality, and intent.

We present here a general discussion of the state of the art in mobile code security. For the sake of brevity, specific details have in places been omitted. Faced with limited space, we have included what we believe is necessary for a fundamental understanding of the technology. We have omitted that which does not contribute significantly to this general comprehension. Springer-Verlag offers an excellent resource for readers interested in learning more.[1]

2 Motivation

Mobile agents can be employed for a wide variety of applications. An oft-cited example of their potential use is in information gathering. A mobile agent could traverse a network of hosts, gather data at each node, and modifying its behavior based upon the information "harvested" at each point. An airfare-finding agent could be programmed with flight criteria (time, price, seating availability, etc.) and then set loose to visit major airline sites. It would return to the

[1] http://link.springer-ny.com/link/service/series/0558/tocs/t1419.htm

client after completing its circuit to present the data it had gathered.

In [4], Fong lists a few of the benefits of mobile agents:

- **Real-time interaction with remote resources:** an agent can interact in realtime with remote resources such as databases, file systems, and sensors.
- **Reduction of communication traffic:** a client can be migrated to a server to carry out lengthy transactions which would be otherwise infeasible over high-cost, high-latency, or unreliable networks.
- **Customization and extension of server capabilities:** agents as application-specific extension code can be downloaded dynamically into server software in order to meet specific client needs.
- **Avoiding distribution of state:** agents avoid the difficulties of maintaining consistency and correctness in client-server applications by localizing computational states.

This new model of distributed computing promises great advances on a variety of fronts: electronic commerce, mobile computing, and information harvesting to name a few. However, to be of practical use in today's Internet, mobile agents and their hosts must deal adequately with a number of diverse security issues.

Consider the classic example of a customer shopping for airline fares. The customer dispatches an agent programmed to search for a flight plan that meets her requirements of departure time, cost, seating, and so forth. The agent first travels to (for example) the Delta Airlines server, gets a quote of $200, and stores this information in its environment. It then migrates to the TaekYorDo Airlines host and with the intention of running a similar query.

Unfortunately, unbeknownst to the consumer and the other airlines, TaekYorDo has their server programmed to act maliciously. It modifies the agent code to return TaekYorDo's quote of $300 to the user as the best fare, no matter what other fares the agent may see in the future. The user gets fleeced and the bad guys get rich.

This is one particular situation that needs to be avoided. Note, however, that the malicious party could have employed any number of other, more subtle attacks: he could scan for the lowest fare and undercut the price by 10%. He could delete fares from the agent's environment, reprogram the agent to crash other servers, or modify the number of seats requested by the consumer to inconvenience both the competition and the consumer.

In the absence of appropriate security measures, a malicious customer could also cause significant problems. She could program her agent to attempt to modify the server state, to carry out a denial of service attack on a particular host, or to attack other agents currently executing on the server.

3 Security Requirements

It is clear that there are a number of inadvertent and deliberate behaviors that are undesirable in mobile code systems. Errant agents should be prevented from modifying server functionality and interfering with the operation of other agents. Renegade hosts should likewise be unable to extract sensitive information, inject malicious code, or

otherwise tamper with the user agents they host.

Such requirements can be grouped into two general classes: host security and agent security. Although the mechanisms of protection that comprise these two classes are, in general, different, they address the same types of threats. In order to be of practical use, mobile code systems should address the following issues [4]:

- **Corruption:** the system should at the very least provide the means to *detect* tampering with code or data. In the case of host security, the system should actively *prevent* tampering.
- **Leakage:** the system should provide the means to protect sensitive data from unauthorized access.
- **Denial of Service:** resource allocation mechanisms should be protected. Malicious entities should not be able to hamper the progress of execution.
- **Masquerading:** systems with strong identification requirements should prevent the "impersonation" of one entity by another.

These threats are familiar in the sense that they have close analogs in traditional operating systems and network applications. There is a stark difference, however, in the context of the threats. Traditional operating systems and network services operate on a limited, well-defined set of interfaces and authentication protocols. The mobile code arena must deal with potentially *arbitrary code of unknown origin executing directly on the local machine*. The context is one in which neither the client nor the server can necessarily be trusted.

4 Host Security

The techniques employed to ensure host integrity in the face of aberrant mobile agents are similar in many respects to traditional kernel and user protection mechanisms. They share a common need to provide protected address spaces to executing processes, resource arbitration to various devices, and to allow the kernel robust control of process execution. However, the necessity of paranoid policies towards mobile agents has given birth to security tools and policies of a much stronger nature than those usually found in more traditional operating systems (Windows, Linux, etc.).

4.1 Corruption

The most common technique used to prevent host corruption is to relegate the visiting agent to a highly restricted execution environment. By restricting the visible name space of executing agents, the host can prevent them from tampering with resources outside of their allocated domain [4].

The most primitive form of such a policy is to deny access to *all* untrusted code. Microsoft's ActiveX controls embody such an all-or-none policy. The user is presented with the identity of the downloaded code and queried as to whether to let it run. In our view, this policy is grossly inadequate. First, once given authorization to execute, the code runs with all the privileges of the user. This places complete trust in the author of the program, a decision that is foolish in light of the preponderance of buggy software and malicious programmers. Second, the authentication method used to determine code identity is poor at best. Users have no actual guarantees as to the functionality of the code; merely the assurance that at some point it passed through a "trusted" authority. There are no formal criteria for code signing

by these authorities and therefore their seal of approval is empty.

The D'Agents system provides an added level of flexibility by distinguishing between *owned* and *anonymous* agents [5]. Trusted (owned) agents are cryptographically[2] authenticated via a network of trusted hosts and allocated privileges commensurate with their level of trust. All other (anonymous) agents are granted severely restricted capabilities on a host-by-host basis. This technique might be appropriate for the closely controlled network where it was implemented (the Department of Computer Science at Dartmouth University), but it fails in the domain of a vast, untrusted network such as the Internet. There is (at this time) no public key infrastructure of the scale necessary to make the scheme viable. In addition, agents that migrate outside of trusted hosts become immediately anonymous, severely limiting their usefulness.

There have been significant steps made with the Java language to address these problems. In their most basic form, downloaded Java applets are executed in a restricted *sandbox* to prevent host tampering. This idea is extended to the arena of mobile agents by the Aglets Object Model [7]. Aglets provide a formal authorization language for specifying the context in which visiting agents may execute. A hierarchy of security policies allows owners of *domains* (e.g., `ucsd.edu`), *contexts* (e.g., an `httpd` process or set of files), and agents to concisely specify privileges. An interesting feature is that the agent itself can specify a preference for its execution. This simplifies authorization of some agents because they are *self-restricting*. It additionally allows groups of agents to cooperate and resources to be matched on a case-by-case basis (subject of course to the policies of enclosing hierarchical groups).

Methodologies based on identity and trust are adequate in some respects but lacking in others. For instance, although you might trust *most* agents from a particular source, all it takes is *one* mistake to ruin host integrity. In light of this, new techniques have been developed to verify the benign nature of the agent code itself.

The basic idea is embodied by the Java *byte-code verifier* [8]. It examines the applet to determine if the code violates security parameters. The Java Virtual Machine (JVM) disallows pointer arithmetic and other potentially problematic operations.

This aspect of Java falls into the broad category of *proof verification*. In [4], Fong enumerates four general classes of such verification proofs: *proof-on-demand*, *proof-carrying code*, *proof delegation*, and *proof linking*.

In proof-on-demand, a host examines the incoming agent, constructs a proof of integrity (in this case, non-malicious behavior), checks its proof, and uses the proof to determine access privileges. The JVM is an example of this protocol. While this is the simplest case, it has the disadvantage of incurring significant overhead on the host machine.

Proof-carrying code avoids this overhead. The creator of an agent publicizes a security policy and then constructs a proof of adherence. This proof is then dispatched with the agent; a host need only check the proof for correctness before allowing

[2] We will not discuss the theory behind digital signatures nor cryptographic authentication in general. An excellent treatment can be found in The Handbook of Applied Cryptography by Menezes, van Oorschot, and Vanstone.

execution. A wonderful example of such a protocol is *typed assembly language* (TAL) [9].

TAL ties formal proofs of code correctness *into the code itself*. An analog in traditional programming languages would be to encode loop invariants into the machine language itself, thereby allowing the hardware to verify correctness on-the-fly. TAL prohibits undesirable memory accesses by rendering them impossible. Unfortunately, formal proofs of code correctness are often very verbose. To be practical, only proof of a subset of interesting cases are usually included.

Proof delegation offloads proof production and verification to trusted hardware. This reduces performance penalties but suffers from the difficulty of propagating such hardware widely enough to be of popular value.

Proof linking also address performance costs by modularizing the proof process. Individual pieces of code are verified separately. If two pieces of code wish to interact then each lists a specific set of requirements that must be adhered to for correctness. The fulfillment of these *obligations* is then checked as needed. This delays the process of verification until the code is actually about to be executed and allows for a finer grain of control over specific cases.

4.2 Leakage

Preventing the "leakage" of sensitive information hosted on the server is closely tied to preventing corruption. Many of the techniques listed above play important roles in protecting data resident on the server.

In the simplest case, agents can be prevented from "stealing" sensitive information by adequately protecting it. The agent cannot steal what it does not have access to.

There are, however, many applications in which it is desirable to grant mobile code access to resources that must not be propagated off the local system. Examples include capabilities for a local display, disk, or sensitive database. In these cases the host might wish to allow the agent to interact with the resource but not to "report" information gleaned from that interaction. For example, one might wish to use a file editor that should be granted access to user files but should at the same time be unable to send ("report") their contents off-site.

An effective technique for dealing with such situations is *history-based access control* [2]. Each agent is assigned is a unique identifier, for example a secure cryptographic hash of its code segment. The host then maintains a persistent state machine for each agent. This automata has as its states various *access classes* that determine the capabilities for an agent in that class and characterize its allowed behavior. Transitions to new states occur when the agent performs certain restricted operations. This mechanism is best illustrated by example.

Consider a host that has access classes of a browser, editor, and shell [2]. Browser agents are allowed to communicate with remote sites, display information on the local display, and create temporary files for personal use. Editors may open, close, modify, and display local files on the screen but must not open any remote connections. A shell may interact with the user, create processes under its direct control, and perhaps have limited access to other local operating system resources (e.g., a sorting function or time-of-day system call).

A visiting agent starts off in a default fail-safe state. If it ever opens a local file it makes a state transition to that of an editor. It is henceforth restricted to the behavioral characteristics associated with that state. It cannot, for example, ever open socket. If, instead of opening a local file, it opens a remote connection (for example, to download a newsfeed), the host moves the agent into the browser state. It thereafter cannot access any local user files. If, in the final case, the agent spawns processes for the user or performs other shell-like operations it is moved into the shell state. At that point it can no longer access local files nor open remote connections.

Compartmentalization of capabilities and association of agents with behavioral profiles ensures the host that the privileges of visiting agents are strictly partitioned.

History-based access control suffers from the same problems as a more general class of agent monitoring schemes. In some cases, it is desirable to provide the agent with a large set of resources in the operating system (or application/language platform) itself. It is inefficient for agents to carry all such code with them at all times. Hence hosts might want to provide agents with *stubs* in place of real system calls. These stubs trap to a monitoring procedure that arbitrates access and emulates or invokes system functionality.

The general technique of *code transformation* can also be included here. Instead of replacing vulnerable systems calls with protected versions, code transformation actually replaces suspicious *agent* code with a "safe" alternative [4].

Such monitoring and replacement schemes can incur significant computational overhead. Providing instrumented (protected) calls to agents is an expensive proposition. Rewriting portions of agent code is similarly costly. However, careful implementation of a smaller subset of operations (e.g., the *Deeds* system [2]) has shown that the cost can be kept within reasonable limits.

4.3 Denial of Service

The techniques used to limit "resource hogging" in traditional operating systems can be extended fairly easily to accommodate mobile code. In local execution, hosts can enforce fair resource allocation by applying "normal" resource sharing policies to agents.

A more interesting question is how to protect a *collection* of hosts from a denial of service attack by mobile agents. For instance, how can a network of hosts prevent the operation of an agent whose sole purpose is to split recursively and migrate to neighboring hosts (a la the famous Internet Worm)?[3]

There doesn't appear to be an easy solution to this problem. D'Agents passes *usage vectors* among trusted hosts to limit this type of behavior [5]. Aglets can be similarly controlled by careful construction of *DomainAuthority* and *ContextMaster* policies [7]. However, the rampant propagation of the Melissa virus illustrates just how hard such enforcement is over a large, heterogeneous and untrusted network like the Internet.

One possible solution is to use "market-based" resource control for mobile agents [1]. In this scheme, users are allocated a finite and unforgable supply of "electronic cash" for mobile agent computation. This supply is invalidated and replenished on regular basis; this prevents "hoarding" while

[3] The distributed equivalent of a C program executing `while (1) fork();` across multiple hosts.

at the same time allowing a reasonable amount of work to be completed in any given slice of time. The user dispatches each agent with a supply "cash" to trade for computation time on the various hosts it encounters. The idea is that malicious agents will "go broke" before they can do much harm.

This technique holds some promise but it is unclear how to apply the system efficiently to large networks such as the Internet. Hosts would be required to adhere to some standardized "pricing" of computational resources. In addition, there would need to be general agreement on the method of distributing such "cash", a prospect we find dim.

4.4 Masquerading

User agents posing as trusted entities could potentially trick hosts into granting them access to protected data and services. *Authentication*, the process of establishing identity, can be carried out according to well-developed cryptographic protocols.[4]

The problem with these protocols is that they are, in general, expensive to implement. Public key based systems require a reliable key distribution infrastructure and have voracious computational appetites (e.g., modular exponentiation of huge numbers). Alternative authentication systems (Kerberos, for example) often have undesirable requirements such as trusted third parties or high levels of interactivity.

Issues of agent identity are tough not for lack of well understood tools but rather because authorization policies are not clear-cut in the mobile arena. *Authorization* is the allocation of access rights once identity has been established. Farmer et al note that it is perhaps *impossible* to determine whether an agent is a valid representative of a user or merely a clone [3]. What are the ramifications of a clone receiving authorization to access protected resources?

It seems necessary that authentication and authorization policies will be tailored to specific host environments. These decisions are, however, hampered by the lack of widespread testing. The jury is still out and is likely to remain so until widespread use of mobile code hones these policies into reliable principles.

5 Agent Security

The hardest question to answer in the mobile arena is how to protect agents executing in foreign and untrusted environments. This question is so hard that until 1997 it was largely held as *impossible* in most cases [3, 10].

The reason this question is so difficult to answer is that it is not well understood. It simply hasn't been encountered before. Traditional operating systems concern themselves with protecting the kernel, system resources, and user processes from one another. Traditionally, *one never writes a program to protect itself from the operating system.* Even the closest analog, fault-tolerance, does not assume an actively hostile kernel. An explicitly untrusted execution environment is something totally new.

5.1 Corruption

Mobile code operates under constant threat of corruption by outside sources. A malicious host could easily tamper with an

[4] Again, see The Handbook of Applied Cryptography for an excellent introduction.

unprotected agent's source code, state information, or accumulated data.

At the very least, users would like to *detect* corruption of their mobile agents. In [11], Yee outlines the following three cryptographic approaches to this problem.

Digital signatures can be used to ensure the integrity of data accumulated during the agent's execution. Each host would be expected to sign any data that it presents. The presumably unforgable nature of these signatures would allow the user to check the integrity and authenticity of any data his agent returns.

A shortcoming of this scheme is that unless the user knows beforehand the identities of the hosts his agent is to visit, he cannot detect the *deletion* of a host's data.

Closely related to digital signatures are Partial Result Authentication Codes (PRACs). PRACs can be used as *forward-secure* digital signatures. Forward security efficiently guarantees that subversion at some time during the agent's execution cannot compromise data gathered before that point.

Although these two tools are useful for the detection of data corruption, they do not allow the user to check whether agent code was correctly executed by the remote host. Yee's third cryptographic solution attempts to tackle this problem. He proposes using certain classes of proofs[5] to allow the user to independently verify that faithful execution was, in fact, carried out.

Although beyond the scope of this paper, the key idea is to offer the user information about the program trace that can be checked probabilistically. Unfortunately, such techniques are at the current time mostly theoretical. It appears that several breakthroughs will be necessary to render them practical.

Trusted coprocessors offer a means to *prevent* host tampering. In the ideal case, the host would provide a secure (tamperproof) hardware environment in which the agent is executed. In the absence of specialized hardware, the host could offer a software arena whose security properties could be publicly verified. The second notion seems harder to achieve than the first [3]; however neither have seen much widespread use so it is hard to draw conclusive observations.

5.2 Leakage

Preventing the host from extracting information from an agent is one of the most exciting topics in mobile systems. Not only will an agent likely wish to prevent disclosure of sensitive data, it might also wish to prevent disclosure of the actual *operation* it is carrying out. In other words, it is not just desirable to protect agent data but *agent executable code as well*.

In the presence of a public-key infrastructure, agents can easily secure the data they gather by requiring hosts to encrypt it to the dispatcher. In the absence of such an infrastructure, securing gathered data becomes much harder. Agents might have to leave sensitive data behind on trusted servers or carry out parallel operations on a limited subset of the data [5].

The most interesting aspect of leakage lies in hiding the agent code itself. *Encrypted functions* allow an agent to carry out operations in such a way that it gives no

[5] Namely, holographic proofs (PCP) and computationally sound proofs (CS).

information to the host as to what function it is computing. Even though the server sees each individual operation it can achieve no understanding of "the big picture". Such functions can be categorized as either *obfuscated* or *algebraic*.[6]

In [6] Hohl lays the framework for obfuscated agents. He propose several methods for reliably rendering agent code and data unintelligible. Agents are obfuscated, timestamped, digitally signed, and then freely dispatched to hostile hosts without fear that they will disclose the algorithm that they are executing. Timestamping is included to forestall hosts from tricking an agent into performing excess work in order to gain information about the operation it is really carrying out. If agents are uniquely obfuscated, trapped agents will "expire" before hostile hosts can "break" them, thereby denying the host anything of immediate usefulness.

Agents utilizing algebraically encrypted functions are described in [10]. Essentially, the agents compute a function *algebraically related to* the one they are really interested in. Drawing heavily on the theory of abstract algebra, the authors show how to "transform" a function of interest into a homomorphism that reveals little about the actual computation. When the user agent returns, the inverse transformation is applied to reveal the desired result.

Unfortunately, encrypted functions are still in their infancy. Algebraic encryption applies only to special classes of functions; it is far from applicable to arbitrary code. Obfuscation lacks formal proofs of security and could lead to an "arms race" of scrambling and de-scrambling technologies.

[6] The nomenclature is ours; it provides a more useful distinction than "encrypted" and "blackbox".

5.3 Denial of Service

There is essentially no way for a mobile agent to compel a host to provide it a service. The only recourse a user has is to enact "rear-guard" measures. In the case of hosts "trapping" agents, a user could keep a migration history in order to avoid interacting with that server in the future [5].

If a user suspects that a host is feeding his agent bad information there is the possibility that dispatching multiple agents and polling them for a consensus could help. However these measures are all reactive; the bottom line is that if a host is unwilling to cooperate the only recourse may be to social or legal pressures.

5.4 Masquerading

The problems posed by host masquerading in the mobile code arena are fairly well understood. Clients can be assured that they are not tricked into releasing data to masquerading hosts by use of strong authentication techniques.

Sensitive information bound for a host can be encrypted with the recipient's public key. In this way, even if the agent is tricked into delivering the information, the attacker must completely compromise the security of the recipient host (by stealing its private key) for it to be of any use.

The widespread employment of Certification Authorities such as Thawte and Verisign in web applications might provide an easy way to realize such a scheme.

6 Conclusion and Future Directions

The field of mobile code security is astonishingly diverse. The sheer number of disparate approaches indicates that this is an

area still undergoing active investigation; academia and industry are yet to agree on which techniques will prove most useful.

Yet some consensus must be achieved before mobile agents become more widespread. Mobile code is potentially too dangerous to unleash without well thought out security policies and effective tools for protecting both agent and host.

Horrendous bungles in client-side scripting security (JavaScript holes, malicious ActiveX controls, etc.) and dangerous active email policies (a la the Melissa virus) are merely the tip of the iceberg. What will happen when similarly widespread technologies can migrate autonomously and carry out data-collecting and reporting tasks?

We have summarized many of the major issues involved with mobile code security. This discussion is by no means exhaustive; limited resources dictated the necessary omission of certain technologies and details. Yet even this general survey of the field brings to light many unanswered questions.

One of the most exciting avenues of inquiry seems to us to be in software-based solutions such as the cryptographic algorithms presented in [10]. Can general algorithms be found to cryptographically secure arbitrary code? Are there other or more efficient techniques that can provide correctness proofs of remote code execution?

We think that it would be interesting to apply traditional cryptographic notions to encrypted functions. For instance, is there an analog for *indistinguishability* (in the sense of a pseudo-random bit generator) in which an obfuscation scheme could be considered secure if its output is indistinguishable from random instructions?

There seems to be healthy debate as to exactly what is and is not possible in the realm of mobile code. It would be interesting to formally evaluate the assertions made in [3].

An area that has received relatively little practical attention is that of using social and legal pressures to ensure security. Are these pressures a viable means by which to compel honesty on the Internet?

Research into general security *policies* for mobile code could likewise prove fruitful. Do we have tools available to implement useful policies? What *are* the useful policies?

Market-based solutions such as [1] might provide an answer. How can they be practically implemented in a medium such as the Internet? Results in these last three areas could potentially be applied to a larger class of problems such as preventing unsolicited bulk email (spam).

A very important question in this era of unique hardware ID's and authenticated user agents is whether anonymity can be preserved. *Should* it be preserved?

Acknowledgments
The authors would like to thank Chez Bob and the manufacturers of Mountain Dew™ brand soda for their assistance in the production of this paper.

References

[1] Jonathan Bredin, David Kotz, and Daniela Rus. *Market based resource control for mobile agents*. Conference on Autonomous Agents 1998.

[2] Guy Edjlali, Anurag Acharya, and Vipin Chaudhary. History-based Access Control for Mobile Code.

[3] 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*, 1996.

[4] Philip W. L. Fong. Viewer's Discretion: Host Security in Mobile Code Systems. November, 1998. `ftp://fas.sfu.ca/pub/cs/techreports/1998/`

[5] Robert S. Gray, David Kotz, George Cybenkos, and Daniela Rus. D'Agents: Security in a multiple-language, mobile-agent system. In *Mobile Agents and Security*, Springer-Verlag, 1998.

[6] Fritz Hohl. Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts. In *Proceedings of the 1997 Workshop on Mobile Agents and Security*, University of Maryland, October 1997.

[7] Gunter Karjoth, Danny B. Lange, and Mitsuru Oshima. A Security Model for Aglets. In *IEEE Internet Computing*, pages 68-76, July/August 1997.

[8] *The Java Language Specification*. The Java language Series. Addison-Wesley, 1996.

[9] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. "From system F to typed assembly language." *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages 1998*, San Diego, CA, January 1998.

[10] Tomas Sander and Christian F. Tschudin. Protecting Mobile Agents Against Malicious Hosts. In *Mobile Agents and Security*, Springer-Verlag, 1998.

[11] Bennet S. Yee. A Sanctuary for Mobile Agents. April 28, 1997. `http://www-cse.ucsd.edu/~bsy/`

Note: Our "core" reading consisted primarily of 2,3,4,5, 7, 10, and 11.