May 1998

Mobile Code Security Techniques

Jonathan T. Moore
University of Pennsylvania

Mobile Code Security Techniques

Abstract

This paper presents a survey of existing techniques for achieving mobile code security, as well as a representative sampling of systems which use them. In particular, the problem domain is divided into two portions: protecting hosts from malicious code; and protecting mobile code from malicious hosts. The discussion of the malicious code problem includes a more in-depth study of the Java security model, as well as touching upon several other systems. The malicious host problem, however, is much more difficult to solve, so our discussion is mostly restricted to ongoing research in that area.

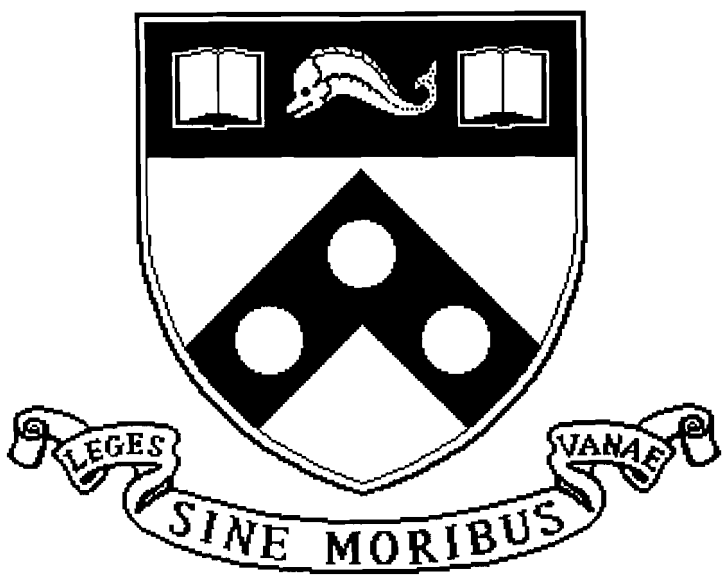
Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-98-28.

Mobile Code Security Techniques

MS-CIS-98-28

Jonathan T. Moore



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

1998

Mobile Code Security Techniques

Jonathan T. Moore
Department of Computer and Information Science
University of Pennsylvania
jonm@dsl.cis.upenn.edu

May 9, 1998

Abstract

This paper presents a survey of existing techniques for achieving mobile code security, as well as a representative sampling of systems which use them. In particular, the problem domain is divided into two portions: protecting hosts from malicious code; and protecting mobile code from malicious hosts. The discussion of the malicious code problem includes a more in-depth study of the Java security model, as well as touching upon several other systems. The malicious host problem, however, is much more difficult to solve, so our discussion is mostly restricted to ongoing research in that area.

1 Introduction

The recent explosion of the Internet, and in particular, the World Wide Web, offers an astounding amount of interconnected computing resources. However, for most users, their use of Internet resources is primarily limited by bandwidth. In particular, especially in home computers, there are many CPU cycles to spare in comparison to the rate at which data can be retrieved.

This suggests that rather than attempting to move data across a network, we might be best served by trying to move applications, which may very well be more compact than the data they operate upon or produce. The usual term for this is *mobile code*—code which is written on one computer is transmitted in some form to a second computer, where it is executed. The two main forms of mobile code that we will discuss here are *applets* and *agents*.

Applets consist of mobile code which is fetched to be executed in a local environment. This technique was popularized by web browsers with embedded Java [GJS96] virtual machines. In particular, it allowed web publishers to reasonably serve interesting "active" content such as animations or games, rather than just static Hypertext Markup Language (HTML) [BLC95] pages or bandwidth-dependent interactive Common Gateway Interface (CGI) [CGI98] content. Here, the user which hosts the executing applet can be viewed as the consumer.

Agents, on the other hand, are mobile programs which are sent out into the network to perform some task for their owner. One of the earliest mobile agent systems was Telescript [TV96], produced by General Magic. Intelligent agents might be able to perform web searches or shop for bargain airline tickets. The idea is that various business would host agent execution environments, and consumers would *produce* the mobile code and send it out into the network.

In either the applet case or the agent case, we have mobile code which is produced by one party and run in an environment controlled by another party. Naturally, this raises some very important questions about the security of mobile code. In the applet case, the consumer would like to execute useful applets while protecting his system from malicious ones. In the agent case, however, the consumer would like to be able to protect his agents from malicious servers. In this paper, we will refer to these two viewpoints as the *malicious code* problem and the *malicious host* problem, respectively.

The remainder of this paper explores the security issues involved with mobile code and surveys existing techniques for addressing them. We will begin with a discussion of the malicious code problem in Section 2, as that is the more well-understood viewpoint of mobile code security. We will then proceed with the malicious host problem in Section 3, studying the current approaches to address this set of concerns. This

paper is not an overview of mobile code systems; we restrict ourselves to the security issues involved and the techniques for solving them. A good overview of mobile code programming languages may be found in [Tho97], and an introduction to the mobile agent paradigm may be found in [Kna96].

2 Malicious Code

The security issues involved with running untrusted code are fairly clear. Consider the most straightforward method of allowing applets: a user downloads an applet in an appropriate binary executable format and simply runs it. Since this would imply that the applet would run with the permissions of the code consumer, the user's system would be quite vulnerable. The applet might be able to randomly write to memory, possibly crashing the machine[1]. Even worse, the applet would be able to read, modify, or even delete the user's private files.

A first attempt might be to require a step of authentication before running an applet. Then the user could be sure only to run code which came from specific trusted sources. This approach however, is unsatisfying in at least two ways. Firstly, not only this would require some sort of public key infrastructure to scale well, but it might severely limit which applets a user can run—even an "untrusted" server might provide useful and benign code. Secondly, and more importantly, though, even code from "trusted" sources might contain bugs which could have malicious (though unintentional) consequences on the user's system.

This would suggest that an ideal solution to the malicious code problem would be one which caught and prevented unsafe actions, whether intentional or not. Then not only could the user feel confident about downloading applets from untrusted sources, but would also have some measure of protection from buggy software. Three different techniques for solving this problem include *safe interpreters*, *fault isolation*, and *code verification*. The next three subsections will address each of these techniques in turn.

2.1 Safe Interpreters

As mentioned above, running straight binaries presents some serious security and safety problems.

[1] Operating systems without address spaces are distressingly common in the home computing environment!

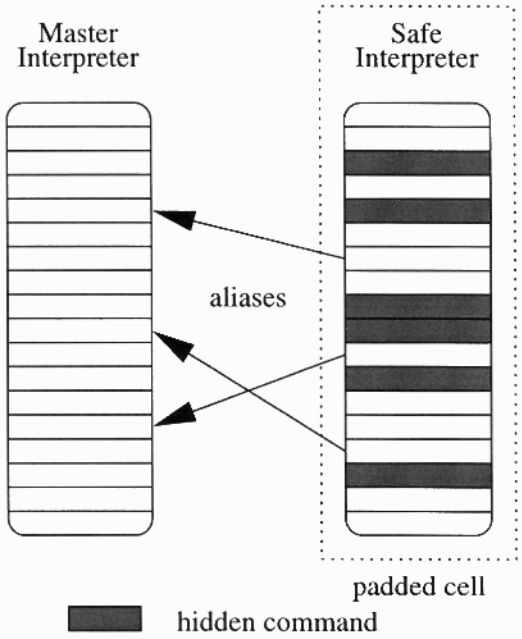


Figure 1: Safe-Tcl architecture [OLW97]

A very common approach to addressing this issue is to forgo compiled executables and instead to interpret the mobile code instead. In this way, the interpreter has fine-grained control over the applet, and can examine each instruction or statement and decide whether to execute it or not. Now the safety of the system is reduced to the correctness of the security policy implemented by the interpreter; a careful code review could provide some measure of confidence to this effect. By contrast, determining whether an arbitrary applet program is "safe" is not decidable, and requiring the user to review all incoming mobile code is certainly not scalable.

Safe-Tcl. One safe interpreter system which illustrates some common security and safety techniques is the Safe-Tcl system [OLW97]. The major construct in use is the *padded cell*, depicted in Figure 1. Each applet is isolated in a safe interpreter where it cannot interact directly with the rest of the application. In turn, the execution environment of the safe interpreter is controlled by a (trusted) master interpreter. This is facilitated by the Tcl language, in which interpreters are first-class values and are highly configurable. In particular, any "unsafe" functions can be *hidden* from the namespace of the padded cell, thus preventing the applet from invoking them.

Of course, if function hiding is used too liberally, it can render an applet not only harmless but also useless! There are certainly cases where the mobile code might need to access disk (*e.g.*, for temporary storage), create a window on the display, or perform some network communication. The Safe-Tcl system uses *aliases* to allow controlled use of unsafe functions. An alias is simply an upcall to the master interpreter which serves to guard some system resource and decide whether to grant or deny the request. The padded cell approach, through proper use of function hiding and aliasing in the construction of the safe interpreters, thus allows multiple applets to be running concurrently, each having its own security policy. Clearly, Safe-Tcl is quite flexible.

However, care must be taken regarding allowing applets to communicate. In their paper describing the system, Ousterhout *et al.* point out that the composition of two sets of safe commands is not necessarily safe itself; *i.e.*, applets can collude to acquire more access than they might individually have been granted[2]. Despite the complexity of the cooperating applet problem, the padded cell approach does simplify the setting of security policies for single applets. Figure 2 shows how this simplification is accomplished; namely, all interaction between trusted and untrusted components of the system occurs through well-defined interfaces (the aliases), so security efforts may be focused there.

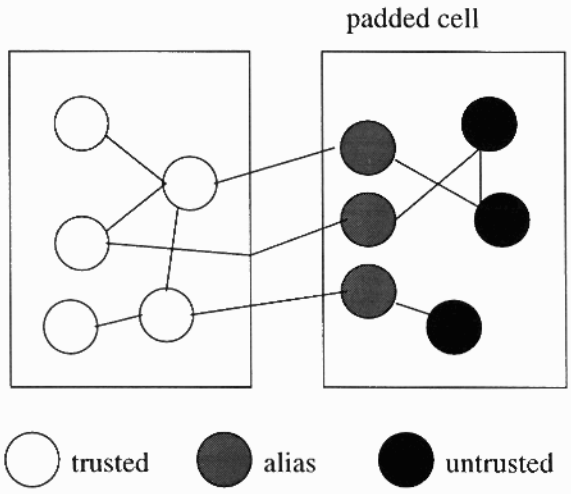


Figure 2: Code interactions in Safe-Tcl [OLW97]

Extensions to Safe-Tcl. As we just mentioned, Safe-Tcl provides an environment which takes care of the safety issues of applets and allows flexibility in the policy for addressing the security issues. Here we mention two projects which use Safe-Tcl and apply their own security policies to solve the malicious code problem.

The first is the Upper Atmospheric Research Collaboratory (UARC) [JRP96], a system designed to allow scientists to collaborate remotely. In UARC, the applications (*e.g.*, test data viewers) are downloaded from a central authority and then executed, allowing simple centralized administration. The main application is called the *browser*, and serves the same role as the master interpreter in the Safe-Tcl architecture; interpreters for different types of applets (including Safe-Tcl itself, Java-enabled Netscape, or Java's appletviewer [GM96]) are then run as subsidiary safe interpreters. The browser abstracts system resources into objects and associates with them access control lists. Access is controlled by using cryptographic authentication to classify the local user, the applet's source machine, and the applet's author. Configurable policies then perform intersections of the various principal's rights to grant the least common access.

A different approach is used by the D'agents [BKR98] system (formerly known as Agent Tcl [Gra96]). Here, the applets are agents are written in Tcl, Java, or Scheme and run in interpreters modified to allow process migration. Resource control is based on an economic system where each agent carries electronic cash and must pay *resource managers* for access to resources. Special processes serve as "banks" which validate transactions and issue the electronic cash. There are additionally "arbiter" processes which are used to discourage cheating on transactions (*i.e.*, receiving service and then not paying for it, or accepting payment and then not rendering the service purchased). Here, each party gives some small amount of "collateral" electronic cash which is forfeited if the other party cries foul, thus discouraging cheating or false accusation. Specifically, these resource managers serve as the master interpreter sides of the aliases in Safe-Tcl. In general, this market-based approach allows the servers to control resource usage by varying their pricing structures and by having the banks limit the outflow of currency to the agents.

[2] For example, an applet granted network access exclusively to hosts inside a firewall might cooperate with an applet allowed only to communicate with hosts outside the firewall, thus effectively bypassing the firewall itself.

Unfortunately, neither of these systems represents a completely general solution to setting security policy for downloaded applets. In the UARC system, for example, usefully empowered applets are restricted to the "official" UARC software—other software is (probably rightfully so) extremely limited in the access control it is granted. Of course, the UARC system is a very specialized application for atmospheric scientists, and although the general architecture could easily be instantiated in other collaborative environments, it is not meant to be a one-size-fits-all application. The D'agents work, on the other hand, seems a little more general; it would be straightforward for a server to set prices on CPU cycles, memory, and disk space based on its current load. One large difficulty seems to be setting a price on read or write access to local files—clearly some useful applets might need this service. It would seem that some sort of authentication would be necessary so that a price could reasonable set for trusted agents or a sale refused to untrusted ones. The paper by Bredin *et al.* does not address this issue; their focus rather seems to be simply allowing machine owners to "rent" their spare computational resources.

Telescript/Odyssey. General Magic was one of the first companies to offer a commercial mobile agents system. Their original system, Telescript [TV96], in light of the overwhelming success of the Java applet platform, has since been reimplemented in Java as is now being marketed as the Odyssey [Whi98] system. Nevertheless, much of the original Telescript paradigm has been preserved.

In Telescript, the notion of security has been included in the design of the object-oriented source language. The class hierarchy includes certain semantic limitations: a class can be *sealed* and thereafter may not be extended or subtyped; also, a class can be *abstract* and thereby not able to be instantiated. In addition, there are operators to turn pointers into *protected references*; an object may not be modified by accessing it through a protected reference. This gives a sort of read-only capability-based feel to the system.

However, one of the main features of the Telescript language is that the security policy for a system may be specified directly in the language. Specifically, each agent (called a *process*) carries with it a *permit* which lists its rights. The permit includes limitations on total lifetime, total memory usage, and CPU priority, but a hosting site may always choose to grant a more restricted set of rights than does the permit. Other rights expressed by the permit include the ability to spawn new processes, the ability to travel to another site, and the abilities to either grant or deny rights to other processes.

In his survey of agent programming languages [Tho97], Thorn points out some weak aspects of the Telescript system:

> The Telescript system includes a number of features to restrict the actions of agents, but they seem to suffer from a lack of systematic design. It is not clear how to be convinced of the consistency of the implemented security restrictions.

One pitfall of having permissions built into the language is that a programming error may lead to a process with a permit that is too permissive. As Thorn puts it: "[Telescript processes] can be hard or impossible to control once launched." Thus, unlike the Safe-Tcl approach, where there is a clear distinction between trusted and untrusted code (recall Figure 2), in Telescript we may have a jumble of agents executing on a given host, all with differing access rights and perhaps interacting in unexpected ways.

Java. Java [GJS96] is perhaps the most well-known applet system in existence. Unfortunately, it is known not only for its security system, but also for the vulnerabilities of that system [Sun98]. We begin by summarizing Fritzinger and Mueller's overview of Java security [FM96].

The sources of Java applets are compiled down into bytecode instructions on the Java Virtual Machine (JVM). An implementation of the JVM is then embedded in an application, *e.g.*, a web browser, which allows the applet to be interpreted. A local *security manager* class is loaded at start-up. All access to unsafe operations must be approved by the security manager. The default restrictions for an applet include: no local disk access; all stand-alone windows created by the applet are clearly labeled "untrusted"; and no network connections to computers other than the server from which the applet was downloaded.

When the bytecodes for the applet arrive at the browser, they are run through a static *verifier*. The verification process confirms that the bytecodes adhere to the Java language specification (*i.e.*, no forging of pointers, class loaders, or security managers). In addition, the verifier checks for violations

of namespace restrictions, stack over- or under-flows, and illegal type casts.

Once the bytecodes pass the verification stage, a *class loader* is invoked which dynamically links them into the namespace. In particular, the class loader keeps separate namespaces for local (trusted) classes and for downloaded (untrusted) classes; this prevents applets from spoofing an existing trusted class. Control is then passed to the bytecode interpreter, and the applet is executed.

The ubiquity of Java applets suggests that this security model is not overly restrictive—useful work can be done this way. However, flaws in JVM implementations of this model, particularly in class loaders and security managers, can be exploited to circumvent the security measures employed. Another shortcoming of this original model is that all applets are given the same set of access rights–there is only one security manager per browser.

An extension of the Java system meant to alleviate this problem is the use of Java Archive (JAR) files. These files are digitally signed by their producers; public key cryptography can then be used to guarantee the origin of the bytecodes, and differing security managers may be used based upon the level of trust placed in the applet's author. JAR files, however, are not in as widespread use as the basic JVM system.

Java extensions. Electric Communities' Trust Manager [Com] is a security framework which allows the specification of the level of trust placed in various principals, and permits rights to be delegated to classes from trusted sources while still retaining the ability to revoke them. They have modified Sun's JVM to allow certificate-based policy decisions that allow for the more granular security control hinted at by JAR files. The changes extend the set of restrictions which can be placed on a downloaded applet, including control over whether other classes can be imported or downloaded and control over the package membership of any downloaded applets.

In [HI97], Hagimont and Ismail proposed an extension to Java and the JVM which adds software capabilities [Lev84] for objects. This would allow mutually-suspicious Java-based mobile agents to control the degree of privilege sharing involved in a cooperative exchange. Their system relies on an infrastructure which will allow agents to mutually authenticate and will permit the granting of an agent's initial permissions when it arrives on a server. Capabilities have been added to Electric Communities' Trust Manager JVM as well as to runtimes produced by JavaSoft [Gol96].

Finally, in [WBDF97], Wallach *et al.* specify three techniques for extending Java's security architecture. Two of these ways, the addition of capabilities and hiding classes *a la* Safe-Tcl, have already been discussed here, so we will simply touch upon the last technique, extended stack introspection. The idea is to require each class to be digitally signed by some principal, which could be a person, the JVM itself, or even another class. When an access request is made for a resource, the established identity is used to access a permissions matrix. Any rights which are granted are then encoded in the execution stack: subsequently called classes then inherit the rights, but when the calling class returns, the rights are popped off the stack, thus preventing a calling class from obtaining the callee's rights. Wallach *et al.* note that the Netscape Communicator 4.0 implements extended stack introspection.

Other interpreted systems. The OCaml [Ler95] programming language implementation has also been used to implement a web browser, MMM. Like Java, applets consist of bytecode files which are dynamically linked and interpreted. However, the security model is quite different from that of the JVM. In particular, the bytecodes are not subjected to the same rigorous verification process. Instead, a crytographic checksum of the interfaces of the downloaded modules is used; the OCaml language is strongly and statically typed, and so a reliance is made on a certified compiler to achieve safety for the system. Security is achieved using familiar techniques: much as the Java class loader maintains a separte namespace for local and remote classes, the OCaml system permits local *module thinning* which results in allowing imported modules to only see a (safe) subset of the locally exported modules' interfaces[3].

Finally, PLAN (Programming Language for Active Packets) [HKM+98] is a language meant to replace network packet headers. Agents here are subject to special requirements specific to their execution environment (*i.e.*, on network routers). Authentication of every packet would be extravagantly costly, and tight control of router resources is very important. The language thus has limited expressibility—there is no direct recursion nor general looping constructs. However, the language *is* sufficiently expressive to write

[3]This is essentially similar to the function hiding capabilities of Safe-Tcl.

programs which run exponentially long in their size, so the system still needs CPU timers and allocation checks. Future work in further language restrictions may permit the removal of these watchdog overheads.

2.2 Fault Isolation

As the reader may perhaps infer from the relative length of the previous subsection, interpreted systems are by far the most common platforms for solving the malicious code problem. However, interpreters suffer a serious performance overhead when contrasted with compiled machine code. Users whose Java applets run achingly slow may wistfully yearn for the ability to safely execute regular binaries. Fortunately, it is possible to move towards this goal using a fairly straightforward method known as *sandboxing* [WLAG93].

Here, the untrusted code is loaded into its own part of the address space known as a *fault domain*[4]. The code is then instrumented to be sure that each load, store, or jump instruction is to an address in the fault domain. This is accomplished in one of two ways:

1. insert a conditional check of the address and raise an exception if it is invalid, or

2. simply overwrite the upper bits of the address to correspond to those of the fault domain,

where the tradeoff is that the former alternative is more useful for debugging but the latter incurs less overhead.

These techniques provide safety at a much lower cost[5] than interpreters. However, we are still subject to security concerns, for which the system shares techniques with the safe interpreters of the previous subsection. Additional instrumentation is done to cause system calls to be turned into calls to *arbitration code*, similar to the aliasing technique of Safe-Tcl. One major drawback of the sandboxing approach is that the downloaded code is no longer platform-independent, which was one of the major design goals for the Java system [FM96].

One additionally relevant technique of software fault isolation can be found in the VINO operating system [SESS96]. Although VINO does not support mobile code *per se*, it does support dynamic kernel extensions and attempts to address the problem of a misbehaving piece of dynamic code. Although the concern here is more one of buggy code, the results would apply to an applet which attempted to hog the resources on its hosting machine.

The kernel extensions (called *grafts*) are run in a sandboxed address space to prevent them from reading or writing inappropriate data or from executing bad instructions. In addition, the grafts are run in the context of a lightweight transaction system. This allows the system to simply kill a graft which is interfering with other processes while still leaving the kernel data structures in a consistent state. Although less important in the applet domain, cooperation among various pieces of mobile code is a key aspect of many mobile agent systems. This technique would allow a malicious agent to be terminated, even if it held a resource like a lock, without leaving shared data in an inconsistent state.

2.3 Code Verification

Although software fault isolation certainly provides mobile code safety with higher performance than interpretation, we are still subject to the overheads of the code instrumentation, as well as the overheads of the indirected calls which access resources. A technique called *proof-carrying code* (PCC) [NL97] can be used to address some of these issues.

Here, the mobile code host decides upon a security policy for an applet. This policy is then codified in the Edinburgh Logical Framework (LF) [HHP93] and published. Now, a burden is placed on the applet author not only to compile the applet to machine code, but also to generate a proof that the code conforms to the conditions specified in the security policy.

Now the code consumer need only verify that the proof supplied is valid and demonstrates that the binary satisfies the security conditions[6], and then simply load and execute the code.

One key question which affects the usefulness of this approach is that of what program properties are expressible and provable in the LF logic used to publish the security policy and encode the proof. PCC has succesfully been applied to minimum and maximum CPU cycle bounds, memory usage and safety, network bandwidth consumption, and type safety. In addition, there is a PCC compiler available for a safe

[4] Also known as a sandbox. The idea is that the untrusted code will only be allowed to "play in its own sandbox."

[5] Wahbe *et al.* found overheads as low as 10-30% over uninstrumented code.

[6] Proof verification is usually far less computationally intensive than proof generation, which may not even be decidable!

subset of C, allowing automatic generation of the safety proofs.

PCC is a very promising approach. The mobile code host can now avoid not only the instruction overhead of sandboxing, but also some of the policy-checking overhead implicit in using the Safe-Tcl alias approach for achieving system security. It does, however, have some drawbacks. Like the basic sandboxing technique, PCC sacrifices platform-independence for performance. In addition, porting is not necessarily straightforward: the LF-encoded security policy and the safety proof must necessarily be closely tied to the operating system and hardware of the machine in question. Nevertheless, the benefits seem to outweigh these disadvantages: PCC is being spun off into a commercial venture, Cedilla Systems [Lee98].

3 Malicious Hosts

Now that we have extensively explored the malicious code problem, let us turn to the converse point of view: the malicious host problem. This problem presents itself primarily in the context of mobile agent programming, where a consumer may have a vested interest in the correct execution of his agent. For example, a shopping agent might carry electronic cash, and it would be undesirable if a host could dupe the agent into paying a high price for some good, or even worse, to simply "mug" the agent and steal its money.

The malicious host problem is daunting indeed; the host certainly needs access to an agent's code and state in order to execute it, so how can sensitive data be kept secret, or how can we guarantee an honest execution of the agent's algorithm? Chess et al. [CGH+95] observe that there are limits to the protection that can be achieved for mobile agents. Firstly, if any portion of an agent's code or state is to be kept private, it must be encrypted. Secondly, we cannot prevent denial-of-service attacks which randomly modify the agent's code or which simply terminate the agent without the assistance of special-purpose trusted hardware.

Therefore, solutions to the malicious host problem should focus on two themes:

1. being able to prove that tampering occurred, and
2. preventing leakage of secret information.

The following subsections outline some current research into this very difficult problem. We will begin with two techniques for the detection of tampering, and will conclude with a theoretical method to preserve secrecy.

3.1 Detecting Tampering

As mentioned earlier, we cannot use technical means to protect our agents from harm. If mobile agent systems existed in a vacuum, it would not seem possible to obtain a satisfactory attempt to solve the malicious host problem, but fortunately, they do not. If we can provably identify a malicious host, then the threat of off-line legal, societal, or physical[7] action would serve to discourage the operators of malicious hosts. Furthermore, it may be that an agent's owner could get some measure of recompense or revenge for the loss of his agent.

The techniques presented in this subsection all rely upon a public key infrastructure to permit the mutual authentication of users, hosts, and/or agents. In particular, since we are interested in *proving* that a host was in fact malicious, the use of digital signatures will be of primary importance.

Execution tracing. Vigna [Vig97] suggests one method to allow tamper detection which involves producing an execution trace of an agent's program. Firstly, the agent's code is divided into two types of instructions: those which depend only upon the agent's internal state, and those whose results depend upon interaction with the evaluation environment. For the former type of instruction, we require the server to record in the trace only the new values of any variables in the agent. For the latter type, however, in addition to recording the new values, the server must digitally sign them.

Once the execution has finished, the server computes a cryptographic hash of the entire trace and returns it to the agent's owner; the hash is in some sense a receipt of the agent's execution. Now, should the agent's owner suspect foul play, he can demand to be shown the trace. The host must then produce the trace, for which the hash value can be verified, and then the trace can be examined to determine if the host either:

1. incorrectly executed an internal-only instruction, or
2. "lied" to the agent during one of its interactions with the environment.

[7] *e.g.*, socks and doorknobs.

However, there are practical problems with this approach. Firstly, this does not alert the agent's owner to any foul play; it merely allows it to be provably identified if the owner's suspicious are raised. Secondly, it places a very high burden on the servers (especially the honest ones), as they must store all of their execution traces in case someone demands them.

Authenticating Partial Results. Yee [Yee97] presents two ways to detect tampering by malicious hosts. The first method involves the use of *partial result authentication codes* (PRACs). An agent is sent out with a set of secret keys $k_1, ..., k_n$. At server i, the agent uses key k_i to sign the result of its execution there, thereby producing a PRAC, and then erases k_i from its state before moving to the next server. This means that a malicious server cannot forge the partial results from previous hops; at worst, it could merely remove them from the agent.

The PRACs should now allow the agent's owner (who also possesses $k_1, ..., k_n$) to automatically cryptographically verify each partial result contained in a returning agent. The property that these messages guarantee is *perfect forward integrity*:

> If a mobile agent visits a sequence of servers $S = s_1, ..., s_n$, and the first malicious server is s_c, the none of the partial results generated at servers s_i, $i < c$, can be forged. [Yee97]

However, if the tampering occurs simply through dishonest interactions with the running agent, this scheme will not automatically detect it. Again, we must rely upon the suspicions of the agent's owner to cause the PRACs to be examined—the PRACs will all be *cryptographically* valid, although one or more may not be *semantically* valid.

Yee presents a speculative approach to detecting this semantic tampering based on *computationally sound proofs* [Mic94]. For a program x, let y be an execution trace for x. Now, the host could send y back to the owner to be verified, but execution traces could be quite large so their transmission may be too costly in terms of bandwidth. Instead, the host can encode y as a *holographic proof* y' that y was the result of running x. This proof y' has the property that the owner needs only examine a few bits of y' to be convinced of its correctness. The server then uses a tree hashing scheme to hash the proof down to a small root value, which is then transmitted back to the owner who gets some confidence that y was correct. The main drawback seems to be the burden placed upon the server. Firstly, the construction of the holographic proof y' is an NP-complete problem, which would seem to make it impractical, particularly if the trace y is already too large to simply transmit back to the owner.

3.2 Preserving Secrecy.

Sander and Tschudin [ST97] present a theoretical result aimed at allowing an agent to preserve some secrecy from the malicious host. The motivation is that there are some situations in which simple detection after-the-fact is insufficient or unsatisfactory. Two examples are when the cost of legal action is greater than the financial loss caused by tampering and when an agent sent to digitally sign something on its owner's behalf has a private key compromised.

Essentially, the problem we would like to solve is the following: our agent's program computes some function f, and the host is willing to compute $f(x)$ for the agent, but the agent wants the host to learn nothing substantive about f. The protocol presented works in the following way, where E is some encryption function:

1. The owner of the agent encrypts f.
2. The owner creates a program $P(E(f))$ which implements $E(f)$ and puts it in the agent.
3. The agent goes to the remote host, where it computes $P(E(f))(x)$, and returns home.
4. The owner decrypts $P(E(f))(x)$ and obtains $f(x)$.

The basic idea is to convert the basic algorithm into a garbled algorithm whose results can only be made sense of by the owner of the agent.

Sander and Tschudin consider representing the function f as a polynomial and then showing that certain classes of *homomorphic encryption schemes* would enable the protocol interaction above. However, there is some question whether a computationally feasible homomorphic encryption function exists: the above protocol would allow an efficient symmetric encryption algorithm with a hardwired secret key to be itself encrypted and sent to a second party. This second party would then be able to use this function to encrypt data without discovering the secret

key, thus effectively providing a public key encryption system. Since there is no known efficient public key algorithm, this suggests that the encrypted algorithm must itself be inefficient (*i.e.* applying E to a function results in nontrivial "code bloat.").

4 Conclusions

The malicious code problem is by far the more well-understood half of mobile code security concerns. The wide-ranging popularity of applet-enabled web browsers alone testifies to the fact that reasonable solutions to this problem exist. Nonetheless, research is actively ongoing to continue analyzing and automating security policies and attempting to remove runtime overhead for enforcement.

The malicious host problem, however, seems to be far less tractable. There are not yet any computationally feasible methods to detect tampering, and even some of the techniques for proving that tampering occurred place a large burden on servers. In addition, it is not clear that it is possible to reasonably provide an agent with any sort of secrecy while it executes in a hostile environment. These problems all would go to explain the lack of widespread use of mobile agents, and in some cases would tend to indicate that secure uses of the agents may be quite limited by a lack of secrecy.

Acknowledgements

I would like to thank Matt Blaze for pointing out clearly the reduction of a public key encryption algorithm to a secret key encryption algorithm in the Sander and Tschudin scheme. Upon closer inspection, this result was in the paper, but I probably would never have seen it had I not known to look for it.

References

[BKR98] J. Bredin, D. Kotz, and D. Rus. Market-based Resource Control for Mobile Agents. In *Proceedings of Autonomous Agents*, May 1998. To appear.

[BLC95] T. Berners-Lee and D. Connolly. Hypertext Markup Language—2.0. RFC 1866, Network Working Group, November 1995.

[CGH+95] D. Chess, B. Grosof, C. Harrison, D. Levine, and C. Parris. Itinerant Agents for Mobile Computing. Research Report RC 20010, IBM Research Division, March 1995.

[CGI98] The CGI Specification. `http://hoohoo.ncsa.uiuc.edu/ cgi/ interface.html`, May 1998.

[Com] Electric Communities. Using the EC Trust Manager to Secure Java. `http://www.communities.com/ company/ papers/ trust`.

[FM96] J. S. Fritzinger and M. Mueller. Java™ Security. Sun Microsystems, Inc., `http://java.sun.com/ security/ whitepaper.ps`, 1996.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[GM96] J. Gosling and H. McGilton. The Java Language Environment: A White Paper, May 1996. Sun Microsystems, Inc., `http://java.sun.com/ docs/ white/ langenv`.

[Gol96] T. Goldstein. The Gateway Security Model in the Java Electronic Commerce Framework. JavaSoft, `http://www.javasoft.com/ products/ commerce/ jecf_gateway.ps`, November 1996.

[Gra96] R. S. Gray. Agent Tcl: A Flexible and Secure Mobile-agent System. In *Proceedings of the 1996 Tcl/Tk Workshop*, pages 9–23, July 1996.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HI97] D. Hagimont and L. Ismail. A Protection Scheme for Mobile Agents on Java. In *Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 215–222, September 1997.

[HKM+98] M. W. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. M. Nettles. PLAN: A Programming Language for Active Networks, April 1998. Submitted to the International Conference on Functional Programming (ICFP'98).

[JRP96] T. Jaeger, A. D. Rubin, and A. Prakash. Building Systems that Flexibly Download Executable Content. In *Proceedings of the 6th USENIX Secuity Symposium*, pages 131–148, June 1996.

[Kna96] F. Knabe. An overview of mobile agent programming. In *Proceedings of the Fifth LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, June 1996.

[Lee98] P. Lee. Proof-Carrying Code. Invited talk, SwitchWare retreat, New Hope, NJ, April 1998.

[Ler95] X. Leroy. Le système caml special light: modules et compilation efficace en caml. Research report 2721, INRIA, November 1995.

[Lev84] H. M. Levy. *Capability-Based Computer Systems.* Digital Press, 1984.

[Mic94] S. Micali. CS Proofs. In *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science*, pages 436–453, November 1994.

[NL97] G. C. Necula and P. Lee. Safe, Untrusted Agents using Proof-Carrying Code. In *Lecutre Notes in Computer Science Special Issue on Mobile Agents*, October 1997.

[OLW97] J. K. Ousterhout, J. Y. Levy, and B. B. Welch. The Safe-Tcl Security Model. Sun Microsystems Laboratories, `http://www.scriptics.com/ people/ john.ousterhout/ safeTcl.ps`, March 1997.

[SESS96] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd ACM Symposium on Operating Systems Design and Implementation*, pages 213–227, October 1996.

[ST97] T. Sander and C. F. Tschudin. Protecting Mobile Agents Against Malicious Hosts. *Lecture Notes in Computer Science on Mobile Agent Security*, November 1997. To appear.

[Sun98] Java Security Frequently Asked Questions (FAQ), 1998. Sun Microsystems, Inc., `http://java.sun.com/ sfaq`.

[Tho97] T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, September 1997.

[TV96] J. Tardo and L. Valente. Mobile Agent Security and Telescript. In *Forty-First IEEE Computer Society Conference (COMPCON)*, 1996.

[Vig97] G. Vigna. Protecting Mobile Agents through Tracing. In *Proceedings of the Third Workshop on Mobile Object Systems*, June 1997.

[WBDF97] D. S. Wallach, D. Balfanzi, D. Dean, and E. W. Felton. Extensible Security Architecture for Java. Technical Report 546-97, Department of Computer Science, Princeton University, April 1997.

[Whi98] J. White. Mobile Agents White Paper. General Magic, `http://www.genmagic.com/ technology/ techwhitepaper.html`, 1998.

[WLAG93] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 203–216, December 1993.

[Yee97] B. Yee. A Sanctuary for Mobile Agents. Technical Report CS97-537, Computer Science Department, University of California in San Diego, April 1997.