

Trust Management and Proof-Carrying Code in Secure Mobile-Code Applications

A Position Paper

Joan Feigenbaum
AT&T Labs
Murray Hill, NJ 07974
jf@research.att.com

Peter Lee
Carnegie Mellon University
Pittsburgh, PA 15213
petel@cs.cmu.edu

DARPA Workshop on Foundations for Secure Mobile Code
March 26–28, 1997

1 Introduction

The popularity of the Java programming language and the concomitant media attention given to the “security holes” that have been found in the Java runtime system have brought the problem of *mobile-code security* to center stage in the computer science research world. In this essay, we describe how the concepts of *trust management* and *proof-carrying code* might be used in mobile-code applications to provide greater security than is afforded by Java and other current technologies. We begin by giving our view of what the mobile-code security problem is and how it both resembles and differs from security problems that existed in the pre-Java world. We then briefly explain two approaches to mobile-code security: *trust management* and *proof-carrying code*. (More complete explanations of these approaches can be found in [1, 6, 7].) Although both of these approaches are only in embryonic stages of development, we believe the early experimental results give some basis for speculating on larger-scale applications. With this in mind, we look at three scenarios in which mobile code may be deployed—programmable network switches, electronic commerce, and agent interactions—and show how trust management and proof-carrying code might be brought to bear.

2 The Mobile-Code Security Problem

The essentials of the mobile-code security problem are these. A *host* computer receives a data object (the “mobile code”¹) from an external source along with

¹Sometimes also referred to as the “executable content.”

a request to execute it in one of the host's execution contexts. The host must decide whether or not to honor the request. Potential reasons not to do so vary from host to host, just as the precise meaning of "computer security" varies widely from context to context. A host may be concerned about a number of potential consequences of executing a particular piece of mobile code, such as:

- damage to the host's file system, *e.g.*, modification or removal either of data files or of already-resident executables
- excessive use of the host's resources, such as cpu time or main memory, to the point that the mobile code effectively disables other processes running on the host
- leakage of private information belonging to the host's established users. Such leakage could be intentional and malicious (*e.g.*, the mobile code could have been written to accomplish theft of information) or it may be unintentional and have unknown consequences (*e.g.*, the mobile code or some other program with which it communicates could have a bug in its crypto module).
- access that the mobile code, once it begins running on the host, may gain to a company intranet or some other private network to which the host is connected

Note that these threats to hosts existed before the popularity of mobile code in general and Java applets in particular. When a person downloads a program from a web page or inserts a floppy disk in the host's disk drive and runs it, often giving it access to all of the host's resources, the same threats are introduced. Some assert that it is the responsibility of the person who runs the downloaded or floppy-disk-transported program to examine it first and to make sure that it will not damage the host. It is our view that this responsibility is more honored in the breach than in the observance and hence that mobile code does not present wholly new security threats.

What is new today is the *scale* of these threats. The promise of Java and other forms of mobile code is that there will be a huge number of mobile programs, performing a huge array of tasks on a wide variety of host types, and created by a huge diversity of programmers (who presumably exhibit a huge diversity of competence and honesty). Thus the semblance (which may not always be the reality) of security provided by shrink-wrapped packages or familiar web sites of a small number of software vendors disappears. Hosts will be receiving programs from diverse and unknown sources and will need to decide whether to trust them.

In response to the need to verify trustworthiness of mobile programs or their sources, one often hears the advice "use digital signatures" or "use authentication." Although signatures, authentication, and other cryptographic building blocks can play a role in securing mobile code, they cannot do the entire job. To authenticate someone or verify his signature, one needs prior knowledge of him. However, mobile programs may be authored by someone of whom a host

has no prior knowledge, and we do not want to rule out the running of such programs. More generally, the signer or source of a mobile program can only “sign off on” properties of the program. He cannot attest to any properties of the host’s environment, and these are presumably crucial inputs to the decision about whether the program should be run in this environment.

3 The Importance of Language in Mobile-Code Security

At its most basic, the problem for the host is to determine whether some set of properties holds for a mobile program that is delivered to it. We refer to these as the “safety properties,” of which there are many different kinds, depending on the application and host requirements. Some properties are assertions about the behavior of the mobile code, for example: The program will only perform actions in the set X , will never perform any action in the set Y , will only perform an action in Z at most W times, and so forth. Other properties say little or nothing about the intrinsic nature of the mobile code but rather address issues of trust and accountability. Examples of this type of property include: The program was authored by X , was inspected by Y who declares that it has property Z , and so on. It is our view that both kinds of properties will be needed in many mobile-code applications.

No matter what kind of properties are required, in order to have a truly secure system it is necessary to *prove*, to some strong degree, that the properties hold. The difficulty is that it is often expensive or even impossible for the host to find a proof from just the code alone. Behavioral properties are often undecidable or intractable, and trust properties often involve the establishment of identity and maintenance of complex trust relationships. This means, at the very least, that the producer of the mobile code must do at least some of the work of finding a proof and then transmit the results of this work to the host along with the code.

In our view, the use of *formal language* is critical in supporting this mode of operation. Since the host and the mobile-code producer must work cooperatively to prove that the code is trustworthy, the host needs a language for specifying the security policy and communicating this specification to all potential code producers. Since the code producer will have to do some of the work of finding a proof, it must have a language for communicating the results of this work to the host in such a way that the host can easily check it for validity. Even the rules for validity checking have to be shared between the host and the code producer, and so this too is best accomplished via a language.

Not surprisingly, both *trust management* and *proof-carrying code* are based on the use of formal language to address the problem of mobile-code security. We now give brief descriptions of these approaches. More complete descriptions are given in [1, 6, 7].

3.1 Proof-Carrying Code

Proof-carrying code (PCC) is a mechanism that supports the construction of easily checkable mathematical proofs of program properties, as well as the formal specification of behavioral safety properties. Using PCC, the code producer is obligated to prove formally that the program has the host-specified safety properties. Once this is done, the proofs can be written out and sent along with the code to the host. The host can then quickly validate with certainty that the proof does indeed establish the specified safety properties. For more information about the applications and principles underlying proof-carrying code, see [6, 7].

PCC works by encoding the proofs in a language based on the typed λ -calculus, called LF [4].² The encoding is performed in such a way that checking the validity of a proof reduces to typechecking in LF. In practice, typechecking LF is fast, and so almost all of the burden of constructing the proof is shifted to the code producer. The code producer has several options for constructing proofs, including use of a theorem prover, a special “certifying compiler,” or even hand-construction of a proof by a human. The host is oblivious to the means by which the proof was obtained (or even whether or not the proof-constructor is correct!), relying only on the LF typechecking for validating the proof. In this sense, PCC attempts to minimize the amount of external trust required in the system: no certifying authorities are needed for proof validation, and the only trusted software in the entire system is the typechecker. The current typechecker is small and easy-to-trust, being written in about 5 pages of C code.

3.2 Trust Management

Trust management is an approach to making cryptographic tools more useful. The most general definition of trust management is *policy-controlled processing of standardized metadata, using trusted third parties*. We will flesh out what this definition means in the mobile-code context. For more information about trust management in other domains, refer to, *e.g.*, [1, 2, 3, 5].

Policy control is the way in which the individual characteristics of the host’s environment are formally encoded in a security policy with which all incoming mobile programs must comply. The trust-management task is to decide whether the metadata, or “credentials,” associated with an incoming mobile program P constitute a proof that P complies with the host’s policy. By *metadata*, we mean anything that is used in the decision-making process besides P itself. Important examples of metadata include digital signatures by trusted parties, public-key “certificates” guaranteeing the validity of signature-verification keys, PCC proofs [6, 7], and information about the path travelled by P before it reached the host. Finally, *trusted third parties* are those whom the host trusts to make certain statements without supplying proofs that they are true. For example, a company C may be in the business of certifying that executable files are virus-free. A host that has had positive experience with C (or has been referred to C by some other host whom it trusts to make this type of

²Actually, a variation on LF, called LF0, is used in our current system.

referral) may regard C 's signature on P as “proof” that P is virus-free. Note that even this straightforward example demonstrates that trust management can be intricate and subtle: The host needs a language in which to express all relevant trust relationships (*e.g.*, C is trusted to certify virus-freedom, but not necessarily to certify correctness or speed), the verification of C 's signature requires an up-to-date copy of C 's public key, etc.

4 Case Studies

For concreteness, we now consider three scenarios involving mobile code.

4.1 Electronic Commerce

The term *electronic commerce* refers to a broad range of applications, some of which may involve mobile code. For example, parties A and B may be long-standing trading partners. (Think of a manufacturer and its regular suppliers or a consumer and her regular service-provider.) In order to automate the transactions they do most often to the fullest extent possible, A may want to permit B to dispatch mobile code that runs on A 's machine, in order to update A 's records, install new releases of relevant software, perform audits, etc. In a very simple model, A may trust B unconditionally and be in a position to exchange and store cryptographic keys securely; thus all A would need to do is check B 's signature (using the copy of B 's public key that it has obtained and stored securely) on an incoming program P and, assuming that the signature is valid, allow any computer under A 's control to host P . More realistically, A may trust B only under certain conditions, *e.g.*, if B is A 's accountant and has been so for years, A may trust B only to dispatch programs that handle accounting functions (but may give B unconditional trust within this accounting domain). In that case, A would require metadata consisting of B 's signature and a proof that P really is an accounting program.

Note that even this relatively simple example involves nontrivial computation: A must be able to encode in a security policy the definition of an “accounting program” and communicate this definition to B . B must then construct a proof (encoded in LF) that the program sent to A is an accounting program. Finally, A must have the ability to validate proofs that programs satisfy the definition.

In general, A will want to trade with many other parties and will not be able to meet all of them in advance and perform “set-up” functions such as the exchange of cryptographic keys. A natural way to accomplish this is to defer trust to a third party C (or several such parties) who performs the role of a “reputation server.” Before running a mobile program P allegedly sent by B , A may require metadata signed by C that provides a valid public key for B , a property S of programs that B is allowed to “sign off on,” and a statement signed by B (and verifiable using the good key obtained from C) that P has property S . Note that the problem of deciding whether P indeed has property S has

been finessed: C has certified not only that B sends around good programs in domain S but also that B can be trusted not to lie about whether a the domain of a particular program actually *is* S . If for some reason C 's certification is not good enough, then again host A will have to insist that B use PCC to attach a proof that its programs are indeed are in domain S .

4.2 Active Networks

The notion of an *active network switch* was proposed by Tennenhouse and Wetherall [8] and is now the basis for a major DARPA research initiative. The basic idea is to allow users to inject programs into the nodes of a network, even the Internet. This would be accomplished by enclosing fragments of mobile code into each network packet, to be executed at each router and switch that the packet encounters on its way to its final destination. Besides allowing users and groups of users to load customized communication protocols into the Internet, active networks seem to provide a means for better support multicast and broadcast applications, essentially by offloading some of the communication-processing workload into the interior nodes of the network.

In a basic sense, the notion of active networks is a return to timeshared computing, and so trust management could be employed to arrange for certifying authorities that provide users with access to particular active network switches. There are two complicating factors, however. First, since the mobile code is enclosed with network packets, the code producers are largely anonymous. Hence, there is little chance that every active network switch in the Internet will be able to exchange cryptographic keys securely with every potential user. Second, there may be a huge number of users, and furthermore the programs that users want to run might be performance-critical. A typical example is a user who uploads into a network switch a program to do translation of video streams to suit the needs of the recipients of the video. In [8], the performance factor leads to a design in which the mobile code actually runs in the unprotected kernel address space, and hence the code must be guaranteed not to corrupt the network switch's operating system. Proof-carrying code addresses these needs by forcing users to prove the safety of their programs before being allowed to execute them on the network switches. Furthermore, the proof-checker is largely oblivious to the language in which the mobile code is written, thereby allowing even hand-coded assembly language to be used.

In special cases in which the PCC proof of a property would take too much time to verify or the desired property is actually unprovable, the switch could demand that the mobile program be signed by a trusted third party.

4.3 Agents

The term *agent* is meant to connote an autonomous program that roams the Internet looking for hosts in which it can run—to its advantage and, in the best of worlds, to the host's advantage as well. Thus an important point about trust management for agent interactions is that it will seldom be the case that trust

decisions can rely heavily on previously acquired information about the agent's author; indeed, it may be inherent in the overall goal of both agent and host that the agent's author be completely unknown to most or all hosts that the agent encounters. This suggests the use of PCC. For some security policies, the proofs enclosed with the mobile programs would be completely self-contained, thereby allowing the ultimate in mobility. If each potential host on the Internet had an LF typechecker and agreed on a common set of safety properties for agents, then agents could be checked for the relevant properties, and the identities and qualifications of their creators would not be needed. The compactness of proof-carrying code and the speed with which PCC proofs can usually be verified make this a very desirable approach when feasible.

Because it will not be feasible to require all potential hosts to agree on an exhaustive set of safety properties, agent-based applications may rely heavily on trusted third parties. The service performed by third party C is to examine an agent P , write an auxiliary program P' the function of which is to probe a host's environment to check whether it has the features needed to run P safely, and digitally sign (P, P') . When a host receives such a pair (P, P') , it checks that the signature was created by one of the C 's in whom it has established trust (and for whom it has a valid public key) and, if this check succeeds, runs the probe P' . Only if P' says "ok to run P " does the host do so. This scenario illustrates an important feature of trust management systems such as PolicyMaker [1] and REFEREE [3]: These systems support *programmable metadata* such as the auxiliary programs P' . Note that it is not only the host that "relies heavily on trusted third parties." Creators of agents will rely on these parties as well: Those who hope to collect money from hosts will have strong incentives to have their agents certified by third parties who are widely known and trusted.

References

- [1] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized Trust Management," in Proceedings of the 17th Symposium on Security and Privacy, IEEE Computer Society Press, Los Alamitos, 1996, pp. 164–173.
- [2] M. Blaze, J. Feigenbaum, P. Resnick, and M. Strauss, "Managing Trust in an Information-Labeling System," to appear in European Transactions on Telecommunications. Available in preprint form as AT&T Technical Report 96.15.1.
- [3] Y.-h. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss, "REFEREE: Trust Management for Web Applications," submitted for conference publication.
- [4] R. Harper, F. Honsell, and G. Plotkin. "A Framework for Defining Logics," Journal of the Association for Computing Machinery, vol. 40, no. 1, January, 1993, 143–184.

- [5] R. Levien, L. McCarthy, and M. Blaze, “Transparent Internet E-mail Security,” <http://www.cs.umass.edu/~lmccarth/crypto/papers/email.ps>
- [6] G. Necula and P. Lee, “Safe Kernel Extensions Without Run-Time Checking,” in Proceedings of the 2nd Symposium on Operating System Design and Implementation (OSDI’96), Seattle, October, 1996, 229–243.
- [7] G. Necula and P. Lee, “Proof-Carrying Code,” Technical Report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University, September, 1996.
- [8] D. Tennenhouse and D. Wetherall, “Towards an Active Network Architecture,” *Computer Communication Review*, vol. 26, no. 2, April, 1996.