Research on Proof-Carrying Code for Mobile-Code Security

A Position Paper

Peter Lee George Necula
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
{petel,necula}@cs.cmu.edu

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

1 Introduction

The advent of the World-Wide Web and the rising popularity of the Java programming language have made the problem of mobile-code security one of the focal points of research in Computer Science today. By allowing code to be installed dynamically and then executed, a host system can provide an flexible means of access to its internal resources. Of course, the idea of installing and executing new software on a system is not new. What is new, however, is the potential for a large number of anonymous agents to use the Internet to deliver an extremely large number of code objects to hosts. Indeed, in some applications, such as the recently proposed "active networks," [7] it is possible for every network packet that is transmitted through a network bridge or router to contain code to be installed and executed in that bridge or router.

There are many problems to be solved before such uses of mobile code can become practical. Of particular interest in this position paper are the following:

- How can the host ensure that the mobile code will not damage it, for example, by corrupting internal host data structures?
- How can the host ensure that the mobile code will not use too many resources (such as CPU, memory, and so forth) or use them for too long a time period?
- How can the host make these assurances without undue effort and deleterious effect on overall system performance?

There are, of course, many other practical problems, such as how to establish accountability and authentication in such large-scale mobile-code environments.

Some of our thoughts regarding these kinds of issues are presented in a separate essay [1]. For this position paper, we will focus on the problem of how to establish guarantees about the intrinsic behavior of mobile programs. In a sense, we will view the problem of mobile-code security as a problem of how to construct efficient and reliable software systems from separate components: we want intimate (that is, efficient) interactions amongst software components (some of which might come from untrusted sources), but in such a way that certain invariants held by each component will not be violated by the others.

Our position is that the theory of programming languages, including formal semantics, type theory, and applications of logic, are critical to solving the mobile-code security problem. To illustrate the possibilities of programming language theory, we will briefly describe one rather extreme but promising example, which is *proof-carrying code* (PCC).

2 Proof-Carrying Code

Proof-Carrying Code is a technique by which a *code consumer* (e.g., host) can verify that code provided by an untrusted *code producer* (e.g., untrusted Internet agent) adheres to a predefined set of safety rules. These rules, also referred to as the *safety policy*, are chosen by the code consumer in such a way that they are sufficient guarantees for safe behavior of programs.

The key idea behind proof-carrying code is that the code producer is required to create a formal *safety proof* that attests to the fact that the code respects the defined safety policy. Then, the code consumer is able to use a simple and fast *proof validator* to check, with certainty, that the proof is valid and hence the foreign code is safe to execute.

Any implementation of PCC must contain at least four elements: (1) a formal specification language used to express the safety policy, (2) a formal semantics of the language used by the untrusted code, usually in the form of a logic relating programs to specifications, (3) a language used to express the proofs, and (4) an algorithm for validating proofs.

In our current implementation the untrusted code is in the form of machine code for the DEC Alpha processor and the specifications are written in first-order logic. As a means of relating machine code to specifications we use a form of Floyd's verification-condition generator that extracts the safety properties of a machine code program as a predicate in first-order logic. This predicate must then be proved by the code producer using axioms and inference rules supplied by the code consumer as part of the safety policy.

Finally, we use the Edinburgh Logical Framework (LF) [2], which is essentially a typed lambda calculus, to encode and check the proofs. The basic tenet of LF is that proofs are represented as expressions and predicates as types. In order to check the validity of a proof we only need to *typecheck* its representation. The proof validator is therefore an LF typechecker, which is not only extremely simple and efficient but independent of the particular safety policy or logic used. This realization of PCC is described in detail in [5].

Note the many instances where elements from logic, type theory and programming language semantics arise in a realization of PCC. Extended use of these formal systems is required in order to be able to make any guarantees about the safety of the approach. In fact, we are able to prove theorems that guarantee the safety of the PCC technique modulo a correct implementation of the LF typecheker and a sound safety policy [6].

2.1 Advantages of proof-carrying code

There are many advantages in using PCC for mobile code. First, although there might be a large amount of effort in establishing and formally proving the safety of the mobile code, almost the entire burden of doing this is shifted to the code producer. The code consumer, on the other hand, has only to perform a fast, simple, and easy-to-trust proof-checking process. The trustworthiness of the proof-checker is an important advantage over approaches that involve the use of complex compilers or interpreters in the code consumer.

Second, the code consumer does not care how the proofs are constructed. In our current experiments, we rely on a theorem prover, but in general there is no reason (except the effort required) that the proofs could not be generated by hand. We also believe that it is feasible to build a *certifying compiler* that builds proofs automatically through the process of compilation. (More about this below.) No matter how the proofs are generated, there is also an important advantage that the consumer does not have to trust the proof-generation process.

Third, PCC programs are "tamperproof," in the sense that any modification (either accidental or malicious) will result in one of three possible outcomes: (1) the proof will no longer be valid (that is, it will no longer typecheck), and so the program will be rejected, (2) the proof will be valid, but will not be a safety proof for the program, and so again the program will be rejected, or (3) the proof will still be valid and will still be a safety proof for the program, despite the modifications. In the third case, even though the behavior of the program might have been changed, the guarantee of safety still holds.

Fourth, no cryptography or trusted third parties are required because PCC is checking intrinsic properties of the code and not its origin. In this sense, PCC programs are "self-certifying." On the other hand, PCC is completely compatible with other approaches to mobile-code security. For example, in another essay [1], we discuss how trust management and PCC can be used together for mobile code security. We also have some experience in using PCC to determine the correctness of applying Software Fault Isolation [3] to network packet filters [4]. In engineering terms, combining approaches leads to different tradeoffs (e.g., less effort required in proof generation at the cost of slower run-time performance) that lead to greater system design flexibility.

Fifth, because the untrusted code is verified statically, before being executed, we not only save execution time but we detect potentially hazardous operations early, thus avoiding the situations when the code consumer must kill the untrusted process after it has acquired resources or modified state.

These five advantages are essentially statements about the advantage of static checking over dynamic checking. We believe that static checking is essential for mobile-code security, and that system designers in general have a somewhat limited view of how static checking can be used.

2.2 Early experience

In order to gain more experience with PCC and to measure its costs we have performed a series of experiments. We started with simple but practical applications such as machine code implementations of network packet filters and the IP checksum routine. For these early experiments the safety policy was focused on fine-grained memory safety. Our packet filters were about 30% to 10 times faster than comparable filters implemented using other approaches, while the safety proofs were smaller than 800 bytes and required no more than 3ms on a DEC Alpha to be validated [4].

We continued our experimentation with more complex safety policies. In one experiment, the "active ping," we write extensions of packet filters that can also allocate and deallocate memory, acquire and release locks and send network packets. In addition to the memory safety we also check that all of the acquired resources are released within a specified time bound. We also put bounds on the quantity of resources used, including the total number of packets sent and total number of instructions executed. For this example, the safety proof increased to 3.5Kbytes and the validation time to 18ms. However we believe that we have only touched the surface of possible proof encoding techniques and much more efficient representations are possible.

2.3 Limitations and current research problems

There are, of course, several serious obstacles to making proof-carrying code a practical approach to mobile-code security. The most basic obstacle is how to generate the proofs. In our current experiments we have used a simple theorem prover, but we have doubts that such an approach will scale to larger and more complex programs. (On the other hand, we have been surprised at our ability to generate proofs with a naive theorem prover for programs such as "active ping," and so perhaps we are being overly pessimistic.) We believe that the notion of a "certifying compiler" will be critical for scaling up the PCC approach, and this is indeed the main focus of our current research. A certifying compiler would generate a proof automatically along with the target code. The idea is to show, in the proof, that certain properties of the source program are preserved in the target program. Since only easily obtained properties (such as type safety) will be compiled and no theorem-proving will be needed, it becomes possible to imagine handling large programs. Note that it is an advantage of the PCC approach that one does not have to trust the certifying compiler, as the recipient of the target programs will always retain the ability to verify quickly and easily the safety of the code.

Another fundamental problem is in the axiomatization of arithmetic. In our current experiments, we have freely added axioms to the theorem prover and the proof-checker, on an as-needed basis. Clearly, this is not an approach that will scale well, and there is a philosophical problem as to whether such ad hoc extension of the logic should be allowed at all. So, it seems that the host will have to be willing do some computation to check the validity of statements about arithmetic that appear in the proofs. The exact nature of such statements, and their impact on the expressiveness of the proofs, the size of the proofs, and on proof-checking time are all key issues that require in-depth development, analysis, and experimentation.

There is also the matter of the language of specifications and the logic used to prove the safety of programs. So far, we have used simple extensions of first-order logic, but it is possible that other logics (such as temporal logic, linear logic, etc.) might be more suitable for practical applications that demand, for example, reasoning about timing constraints. Again, a great deal of research and experience is needed here.

Finally, we see a basic need for a great deal more experimentation, even with the system and logics that we have already developed. Basic questions about how big the proofs get in practice, and whether some scaling up can be achieved by modular construction of proofs and programs, have yet to be answered in an experimental framework. We strongly encourage further research into such experimentation.

3 Summary

It is our position that the theory of programming languages, which we take to include formal semantics, type theory, and logic, provides methods and systems that will be critical to achieving a high level of security in mobile-code applications. We have briefly described one approach, proof-carrying code, that illustrates the potential of programming-language theory in this arena, essentially through the exploitation of static checking. While there are still many difficult research problems to be solved, we believe that the past and current results show enough potential to warrant a great deal of further work.

References

[1] J. Feigenbaum and P. Lee. Trust management and proof-carrying code in secure mobile-code applications (A position paper). Submitted to the *DARPA Workshop on Foundations for Secure Mobile Code*, Monterey, California, March, 1997.

[2] 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.

[3] R. Wahbe and S. Lucco and T. E. Anderson and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles,* December, 1993, 203–216.

[4] 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.

[5] G. Necula and P. Lee. *Proof-Carrying Code.* Technical Report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University, September, 1996.

[6] G. Necula. Proof-carrying code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97),* Paris, January, 1997.

[7] D. Tennenhouse and D. Wetherall. Towards an active network architecture. *Computer Communication Review*, vol. 26, no. 2, April, 1996.