Infrastructure for Proof-Referencing Code

Carl A. Gunter
gunter@cis.upenn.edu

Peter Homeier
homeier@cs.ucla.edu

Scott Nettles
nettles@cis.upenn.edu

February 26, 1997

Abstract

We discuss ideas for using the Higher-Order Logic (HOL) theorem-proving system as an infrastructure for programs that reference or carry proofs of their correctness. Such programs, which we call *Proof-Referencing Code* (PRC), could be useful or even essential for applications where security of mobile code is important, but where authentication is impractical and runtime checking is expensive. We propose an experiment to determine if PRC can be used to provide a flexible approach to providing security and performance in a more general context than has been shown before. Our goal is to develop a new kind of runtime system based on PRC.

1 Trust but Verify

A key collection of trade-offs for mobile code concerns the over-head involved in locally executing programs that are potentially untrusted. There are three possible approaches: (1) trust anyone, (2) trust only your friends, and (3) (trust but) verify. In general the first option will make sense only when a community is small (for instance, users in a single administrative domain) or cases where trust is not really an issue because the capabilities supplied to the mobile program by the host are too minimal to generate any security concerns (postscript for printers is an example of this, 'ping' is a more debatable example, especially given recent events). Hence the interesting cases are the second and third ones. The second approach will typically involve some form of authentication fashioned from cryptographic techniques. There are numerous impediments to using this approach, such as government regulations on the export of security protocols (limiting international use) and proprietary claims to some of the best-tested techniques (leading to burdensome contract negotiations over royalties). This suggests that the third approach, verification, is an important avenue to explore.

One major advantage to verification as a way to ensure safety of mobile code arises from the fact that the basic technology for doing this has been considerably developed in other contexts. In particular, one may draw on experience from operating systems, some of which provide memory protection which verifies at a very low level that certain memory accesses are safe, and from programming languages, where type systems provide safety guarantees. The second of these is especially well illustrated in the design philosophy of the Java programming language [3], which runs on a virtual machine [7] that employs a dynamic 'verifier' to enforce host security policies for the execution of the compiled bytecode of web applets. This technology is supported by advances in runtime systems (especially garbage collection) and the specification of programming languages (providing precise machine-independent semantic descriptions).

2 Static or Dynamic?

The design space where one assumes a 'verification' philosophy offers two principal alternatives: (1) ensure that the program is safe before running it and (2) run the program but check its actions to see that none are unsafe. Let us call these approaches *static* and *dynamic* checking. Each is widely used. For instance, OS memory protection is based on dynamic checking, whereas many programming languages rely on static type-checking to provide runtime behavior guarantees. An extreme case is the SML programming language [8], whose origins as a theorem-proving meta-language were based on static type safety of that language. On the other hand, most Lisp-family programming languages rely on dynamic checks to ensure controlled errors on type-incorrect programs.

The tradeoff between static and dynamic checking is driven by efficiency and information. Since more is known at runtime (for instance, the actual values passed in a procedure call), it is possible to carry out a more detailed check of safety on the known arguments, but this check may need to be done many times. If the needed information could be obtained before running the program, a single check at that time might save repeated runtime checks. Here's a family metaphor: a child in a child-safe room requires less supervision than one in an child-unsafe room. Hence having a child-safe room can enhance efficiency by reducing the effort devoted to supervision. However, it can be difficult or impossible to make a room child-safe, so monitoring may be necessary or less costly.

3 Verifying More

One of the drawbacks to the existing technologies for static and dynamic verification coming from operating systems and programming languages is that these techniques are limited to a small range of properties they are actually capable of ensuring. Operating systems offer memory protection (or at least the reliable ones do). Programming languages can go further and support certain kinds of data abstraction by controlling name spaces. However, neither approach is adequate to ensure all of the properties involved in general program correctness. In general a specification will require a variety of properties, some of which are beyond expression

as types in typical programming languages, or as memory protection in any OS.

Our work on active networks at Penn has led us to consider ways in which it may be possible to move beyond these boundaries. There is a single technical insight that we would like to use to advance both of these approaches. The insight is this: *it is sometimes easier to see that an answer is correct than it is to produce a correct answer in the first place.* For a good metaphor: it is much easier to see that a jigsaw puzzle is solved than it is to solve it. This translates into two strategies for providing better verification assurances to hosts that might run mobile code. First, if it is known that a program is intended to compute a particular thing, then the program can be run and the answer checked. This is called *program checking* [1] (but *dynamic* verification might be a better name for it). Second, if a proof that a program computes a particular thing is cited, then that proof can be checked. We call this called *proof referencing*. These approaches are complementary.

To see a very simple example of program checking, suppose a program is meant to compute the square root of x within a given accuracy. To achieve program checking for this program, just square its outputs and check to see if they are within the desired range. This idea can be extended to much more interesting applications. However, program checking has two drawbacks. First, it is only able to check that an answer is correct: it does not, by itself, show that a program *will* produce a correct answer, only that it has not *yet* in execution produced an incorrect one. Indeed, a dynamically-checked program is no more reliable than the less-reliable of the checked program and the checker. So, if the correctness of the checker itself is in doubt, then little constructive is contributed by using it. Second, the checker suffers from the usual problem of runtime checking: it must be reasonably efficient or it will degrade the performance of the program it is checking. For instance, if a procedure produces many values and each one must be checked, this could be quite burdensome.

4 Proof-Referencing Code

The alternative to dynamic checking is generally called *program verification*, an endeavor with a history of disappointments. Basically, most programs are too complicated to verify completely; most of the insights about program properties that went into writing the program are not present in the code itself, so autonomous theorem-proving systems are unlikely to be able to find correctness proofs. Interactive proofs require a skilled person to construct the proof of correctness, usually with modest automated support. The result is that it is basically out of the question to insist that a host check that a mobile program is correct or safe by static means.

Various projects have attempted to attack this problem. For instance, 'inferential programming' [11] advocated using an environment and method for collecting information from the programmer as the program was written. Another idea is that of a *verification condition generator*, which uses programmer annotations to generate conditions for verifying a program. A culmination of these and similar ideas was discussed recently under the names 'self-certified code' and 'proof-carrying code' by Necula and Lee [10, 9]. The idea is another variant on the principal that it may be easier to check an answer than it is to produce it. For a mobile program, it is the creator of the program who knows the key reasons it is correct, not the host that receives the program. Hence it is reasonable to shift the burden of proof onto the supplier of the mobile program. The mobile program is paired with a proof of its safety and delivered to a host. It is easy for a computer to check a formal proof, even when the proof may have been very difficult to create, so the host checks the proof and runs the program.

While we don't want to add any confusion by differing from the terminology introduced by Necula and Lee, we'd like to refer to this technique as *Proof-Referencing Code (PRC)* to be more general about what the code must contain. To see the point, let us discuss some pragmatics.

5 Needed Infrastructure

The aim of [10] was to show that a packet filter can be made faster by proving to an OS kernel that the filter would respect its memory protection requirements, thereby enabling the omission of expensive runtime safety checks. This demonstration is useful in bringing up a variety of issues that would arise in any context that sought to use these methods in a more routine manner. First, there must be a common understanding between the host and the code provider about the logic in which the desired property is being proved. It would, of course, be pointless for the host to use a proof-checker supplied by the code supplier. Second, the proof of any non-trivial property of a program probably relies on a great deal of basic mathematics. It is impractical to assume that the host is aware of anything that may be needed, since it may have been developed by the code supplier. It is also impractical for the code supplier to send a large proof that develops a relevant branch of mathematics as part of every mobile program.

From these two facts we can conclude that standardization and infrastructure will be crucial to the practical use of PRC. But what *kind* of infrastructure is required for PRC? First, it is necessary to have a suitable logic. This logic must also have some standardized form for its proofs so they can be checked by a host. It is probably best to use a collection of logics and achieve interoperability by some kind of common embedding into a general-purpose logic of which they can all be viewed as sub-sets. Second, the logic must have a way of building libraries of theorems that can be referenced remotely. This will allow PRC's to refer to the libraries for basic mathematics needed as a foundation for their more code-specific proofs. Third, the logic must be capable of talking about properties of the code language. Fourth—the most interesting criterion—it must be usable by the system that compiles or evaluates the code.

6 Higher-Order Logic System and Theories

Church's higher-order logic was used as the basis for a goal-directed theorem-proving system, HOL, by Michael Gordon [2]. The HOL system was originally used for hardware verification, but numerous projects have shown that it is also useful for software. HOL has an international user community and a large base of theories (HOL terminology for a library of theorems) derived by its users over the years. HOL is based on a short list of basic axioms; the tens of thousands of lines of facts in its world-wide libraries have all been proved from these axioms using the HOL system. One particular feature of this way of doing things is that any user will have as much confidence in the theories of another user

as they have confidence in the *HOL system* (as opposed to the author of the theory).

HOL libraries could form the basis of a PRC infrastructure. We plan to experiment with this idea by an exploration based on work of Peter Homeier [4, 6, 5] which has examined the correctness of a *verification condition generator* (VCG) for a small imperative programming language. A VCG processes programs written in the specified language, internally constructs a proof of the program's correctness, and produces as its result a set of lemmas called *verification conditions*, as the remainder left for the programmer to prove. He was able to show how to prove the soundness of the verification condition generator in HOL, that for all programs submitted to the VCG, the truth of the verification conditions that it produced in fact guarranteed the total correctness of the program with respect to its specification. Such a proof establishes with security many of the connections we would need to have between HOL and the programming language in a PRC system, in particular the fundamental connection that program proofs imply verified code. This kind of VCG would make PRC more practical, by reducing both the volume and the complexity of the proofs to be attached to PRC.

7 A Compiler and Runtime System for PRC

Our principal idea is to develop a compiler and runtime system whose dynamic checking is inversely proportional to how much the compiler knows about the program. This, by itself, is nothing new: compilers typically attempt to gather information about programs in order to reduce dynamic checking. However, the kind of information they must settle for is somewhat limited. Either this must be entirely inferred by the compiler, or it must be guided by some programmer pragmas, but no system we are aware of can accept a *general mathematical proof* of a property as an input that achieves an (automatic) optimization.

Our specific goal is the following. We would like to extend the language of [5] to include arrays. Then we will develop a suitable format for PRC that includes proofs about array bounds access. This will then be used in a distributed system in which a PRC producer can send a proof to a PRC host. At the PRC host the proofs provided by the producer will be checked and, if they are correct, used to optimize the compilation and runtime checking of the program part of the PRC. As part of this system, we will also develop a system that will allow us to store HOL theories so that remote references in PRC can be satisfied. We hope to show through this experiment a significant speedup in execution, through the safe and secure elimination of run-time array bounds checking.

One intriguing aspect of this design is that it provides a novel kind of performance-tuning knob for mobile code systems. Code that carries no information about its safety properties will be heavily checked dynamically. For example, like current packet filters, it might be interpreted so that the results of each instruction can be verified. This would be appropriate for prototype code, or code that is infrequently executed. To improve performance, the programmer can provide more information about code, allowing the host to omit the dynamic checks. In general, this will give us the ability to trade off between the amount of work the programmer does in developing the code, and the amount of work the system does in executing it, while avoiding compromising safety.

8 Applications to Active Networks

Our interest in secure mobile code arises from our work on active networks. An *active network* is one based on a 'store, compute, and forward' model in which the switch is programmable, allowing customizable forms of communication. For instance, a message may contain a program that is executed by the switch to facilitate an application-specific routing strategy. This contrasts with the protocol of the Internet, which is based on a 'store and forward' model of computing: an Internet switch receives a message and deals with it based on the destination to which it is addressed.

There are many potential applications of programmable switches based on various resources they could provide to users and operators. Perhaps their most compelling advantage is that the evolution of an active network will rely less on standards adoption and can therefore proceed at a pace determined by technology rather than standards committees. However, programmability of switches brings up numerous questions about security and the fair allocation of resources. Much research on active networks is currently focused on finding a design to provide good functionality while dealing sensibly with these problems.

The programs that are used to provide customized communication on switches are a form of mobile code that we call *switchlets*. Switchlets could benefit significantly from PRC techniques if PRC allows switchlets to run safely on shared switches without authentication and with protection boundaries that are not expensive to enforce. It is our hope that this technology can be developed to a degree that it can play a significant role in an active network context combining suitable degrees of mutual trust, trust of friends, and trust with verification.

References

[1] Manuel Blum and Sampath Kannan. Designing Programs that Check Their Work. *Journal of the ACM*, 42, 1995.

[2] Michael J.C. Gordon and Tom F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[3] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[4] P. V. Homeier and D. F. Martin. A Mechanically Verified Verification Condition Generator. *The Computer Journal*, 38(2):131–141, July 1995.

[5] Peter V. Homeier. *Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures*. PhD thesis, University of California, Los Angeles, June 1995.

[6] Peter V. Homeier and David F. Martin. Mechanical Verification of Mutually Recursive Procedures. In M. A. McRobbie and J. K. Slaney, editors, *Proceedings of the 13th International Conference on Automated Deduction (CADE-13)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 201–215. Springer-Verlag, July 1996.

[7] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[8] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[9] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM Press, 1997.

[10] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Second Symposium on Operating System Design and Implementation (OSDI '96)*, 1996.

[11] William L. Scherlis and Dana S. Scott. First steps towards inferential programming. In R. E. A. Mason, editor, *Information Processing 83*, pages 199–212, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).