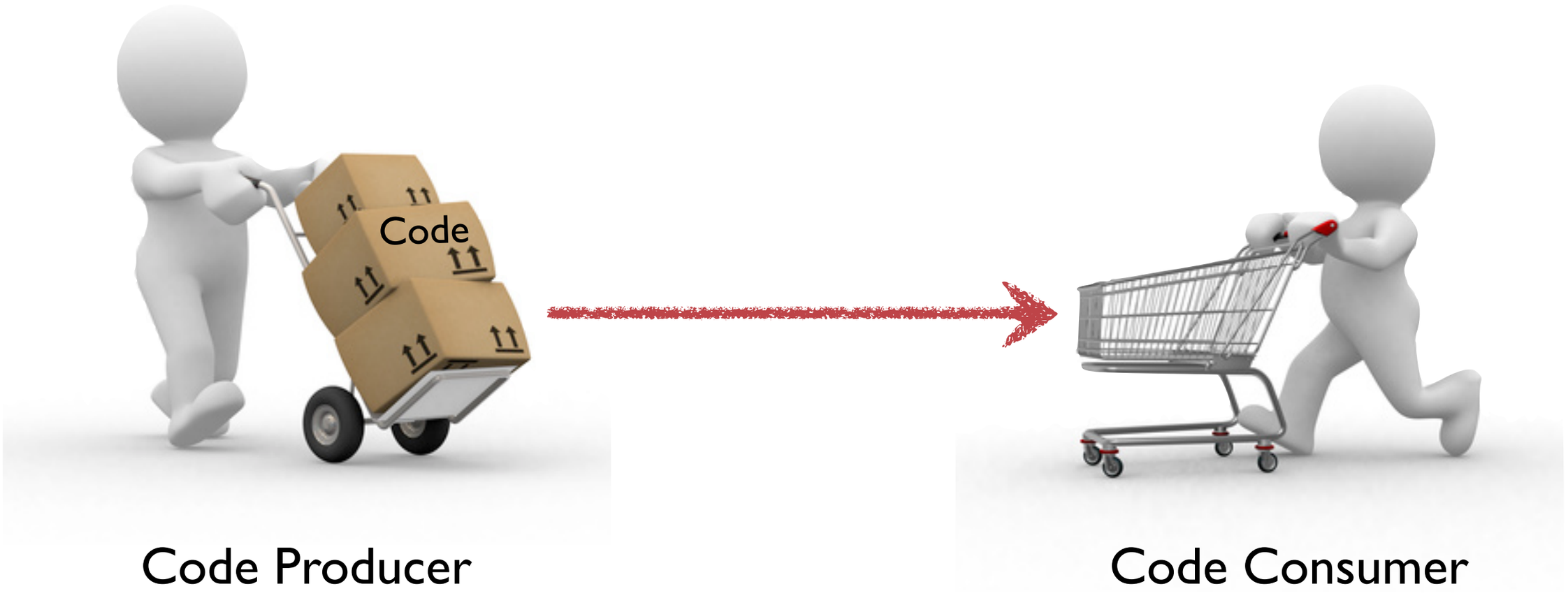
Introduction to Proof-Carrying Code

Soonho Kong

Programming Research Lab.
Seoul National University
soon@ropas.snu.ac.kr

7 August 2009
ROPAS Show & Tell

Proof-Carrying Code



Code Producer

Code Consumer

Proof-Carrying Code

Code Consumer

Proof-Carrying Code

Code Consumer

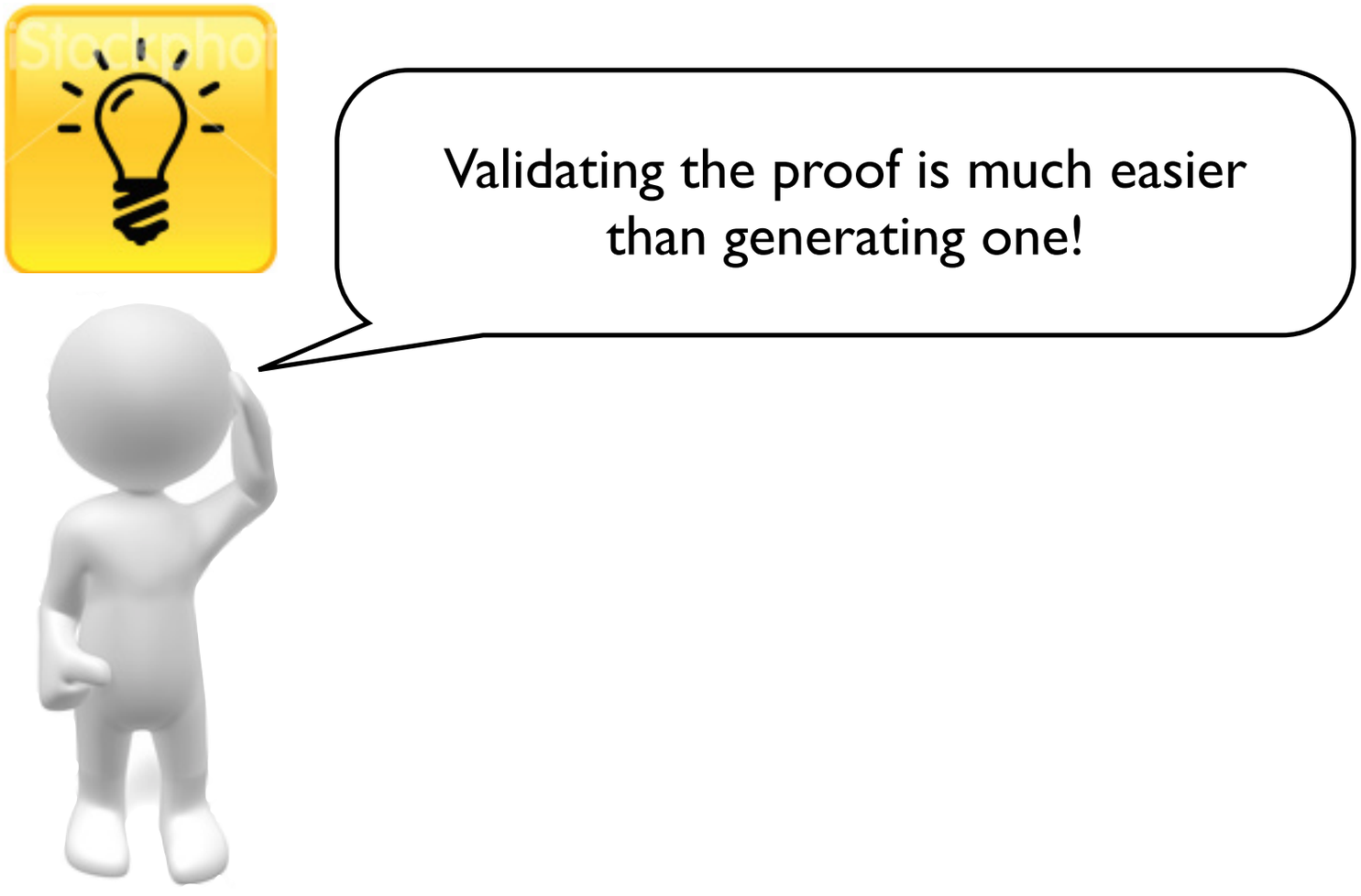
Proof-Carrying Code

Code Consumer

Proof-Carrying Code

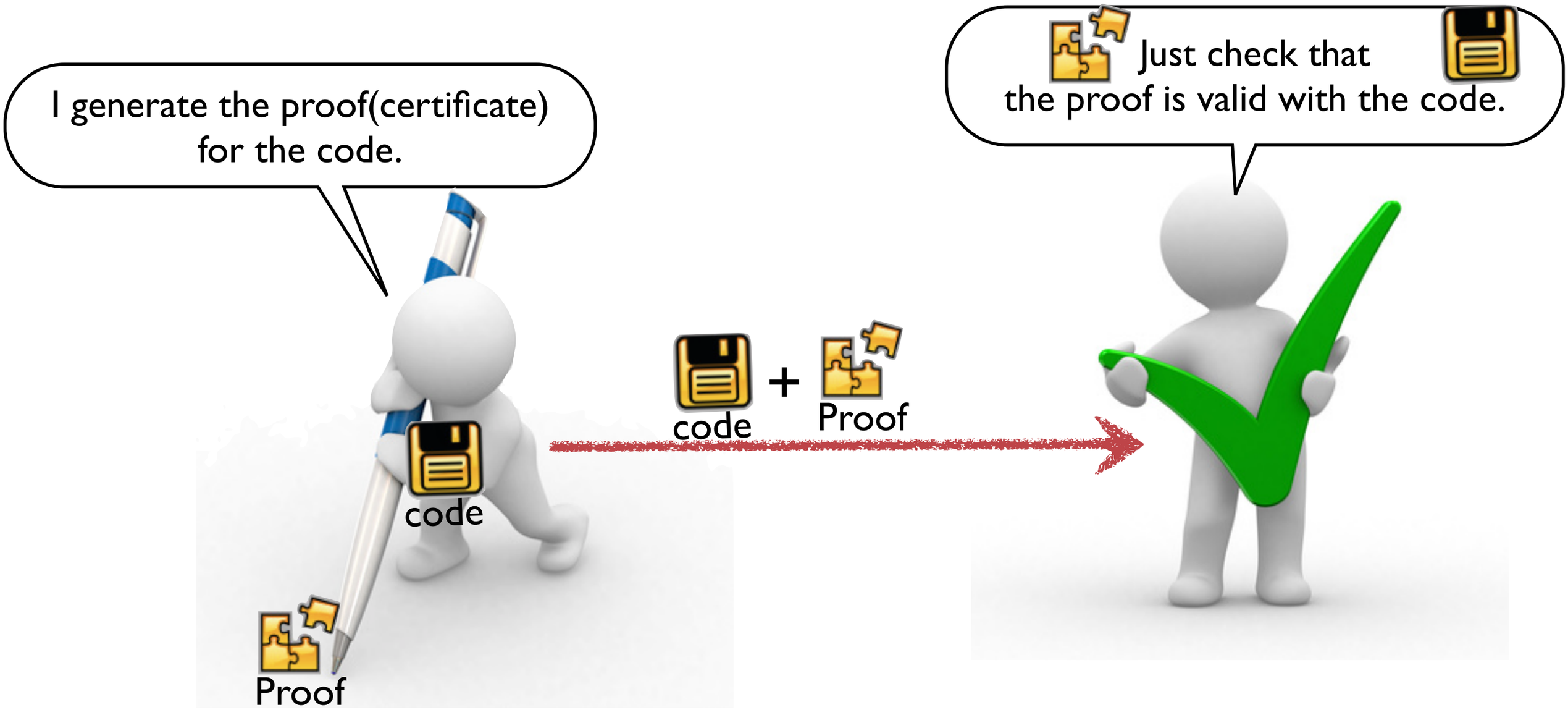
Code Producer

Proof-Carrying Code



Code Producer

Proof-Carrying Code



Code Producer

Code Consumer

Proof-Carrying Code

George C. Necula

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213–3891

necula@cs.cmu.edu

Abstract

This paper describes *proof-carrying code* (PCC), a mechanism by which a host system can determine with certainty that it is safe to execute a program supplied (possibly in binary form) by an untrusted source. For this to be possible, the untrusted code producer must supply with the code a *safety proof* that attests to the code's adherence to a previously defined safety policy. The host can then easily and quickly validate the proof without using cryptography and without consulting any external agents.

In order to gain preliminary experience with PCC, we have performed several case studies. We show in this paper how proof-carrying code might be used to develop safe assembly-language extensions of ML programs. In the context of this case study, we present and prove the adequacy of concrete representations for the safety policy, the safety proofs, and the proof validation. Finally, we briefly discuss how we use proof-carrying code to develop network packet filters that are faster than similar filters developed using other techniques and are formally guaranteed to be safe with respect to a given operating system safety policy.

1 Introduction

High-level programming languages are designed and implemented with the assumption of a closed world. Taking ML as an example, the programmer must normally assume that all components of the program are written in ML in order to establish that the program will have the properties conferred by type safety. In practice, however, programs often have some components written in ML and others in a different language (perhaps C or even assembly language). In such situations, we lose the guarantees provided by the design of ML unless extremely expensive mechanisms (such as sockets and processes) are employed. In implementation terms, it is extremely difficult to determine whether the invariants of the ML heap will be respected by the foreign code, and so we must use some kind of expensive firewall or simply live dangerously.

This problem is exacerbated in the realms of distributed and web computing, particularly when mobile code is allowed. In this kind of situation, agent A on one part of the network might write a component of the software system in ML, compile it to native machine code, and then transmit it to an agent B on another node for execution. How does agent A convince agent B that the native code has the type-safety properties shared by all ML programs, and furthermore that it respects the representation invariants chosen for maintaining the state of B's heap?

There are many other manifestations of the same problem. For example, in the realm of operating systems, it is often profitable to allow application programs to run within the same address space as the operating-system kernel. Once again, the problem is how can the kernel know that the inherently untrusted application code respects the kernel's internal invariants. The problem here seems even worse in practice, because the kinds of properties required of the application code are difficult in the sense that standard type systems cannot express them easily. For example, in the SPIN kernel [1], there are often basic requirements about the proper use of synchronization locks that would be hard, if not impossible, to express in the ML or Modula-3 type systems.

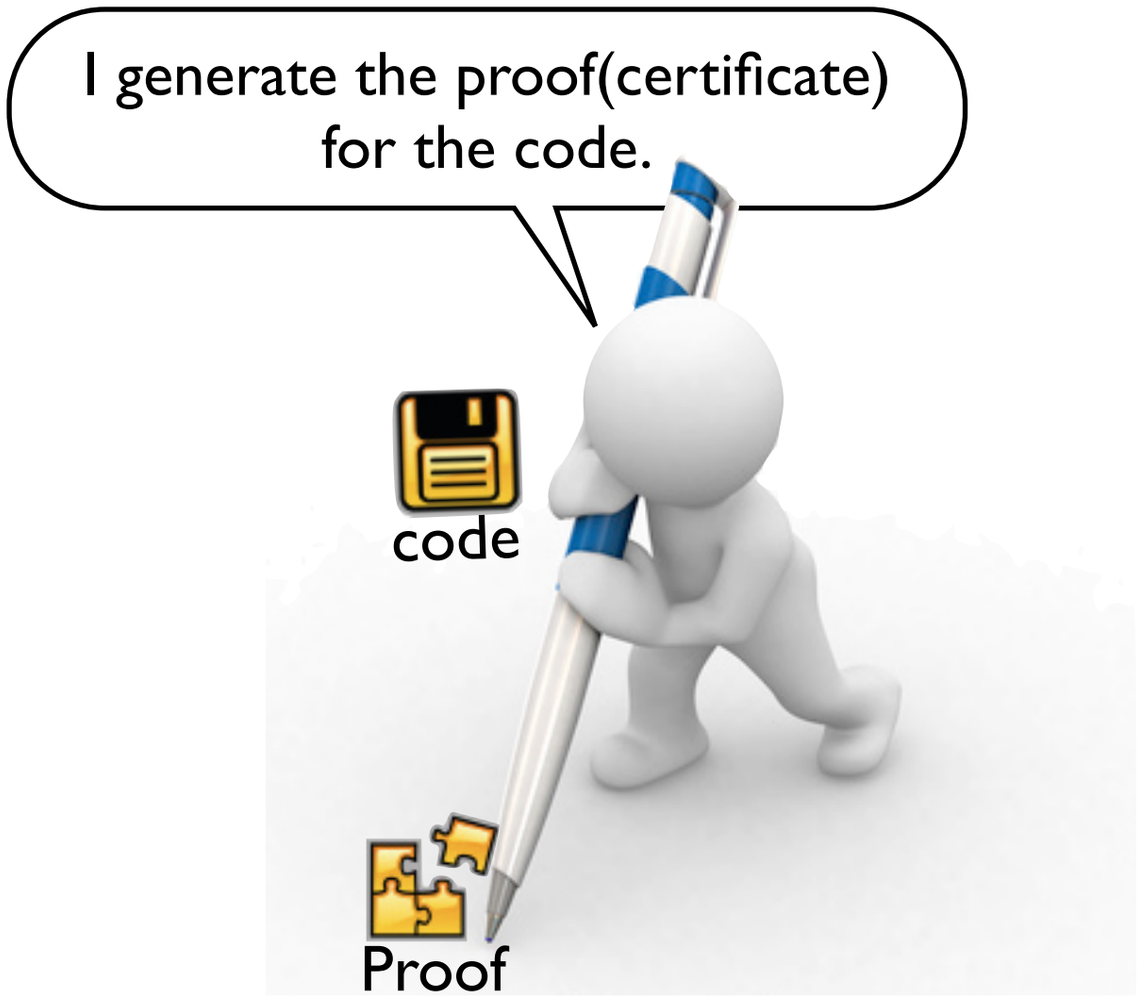
This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software," ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

To appear in the Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97), Paris, France, January 15–17, 1997.

Issues in Proof-Carrying Code

There are several fundamental issues for PCC to be practical.

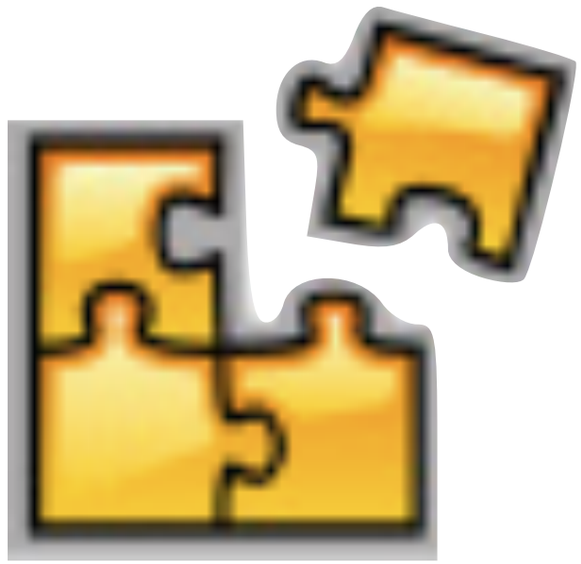
Issues in Proof-Carrying Code



Code Producer

1. The proof needs to be generated automatically.

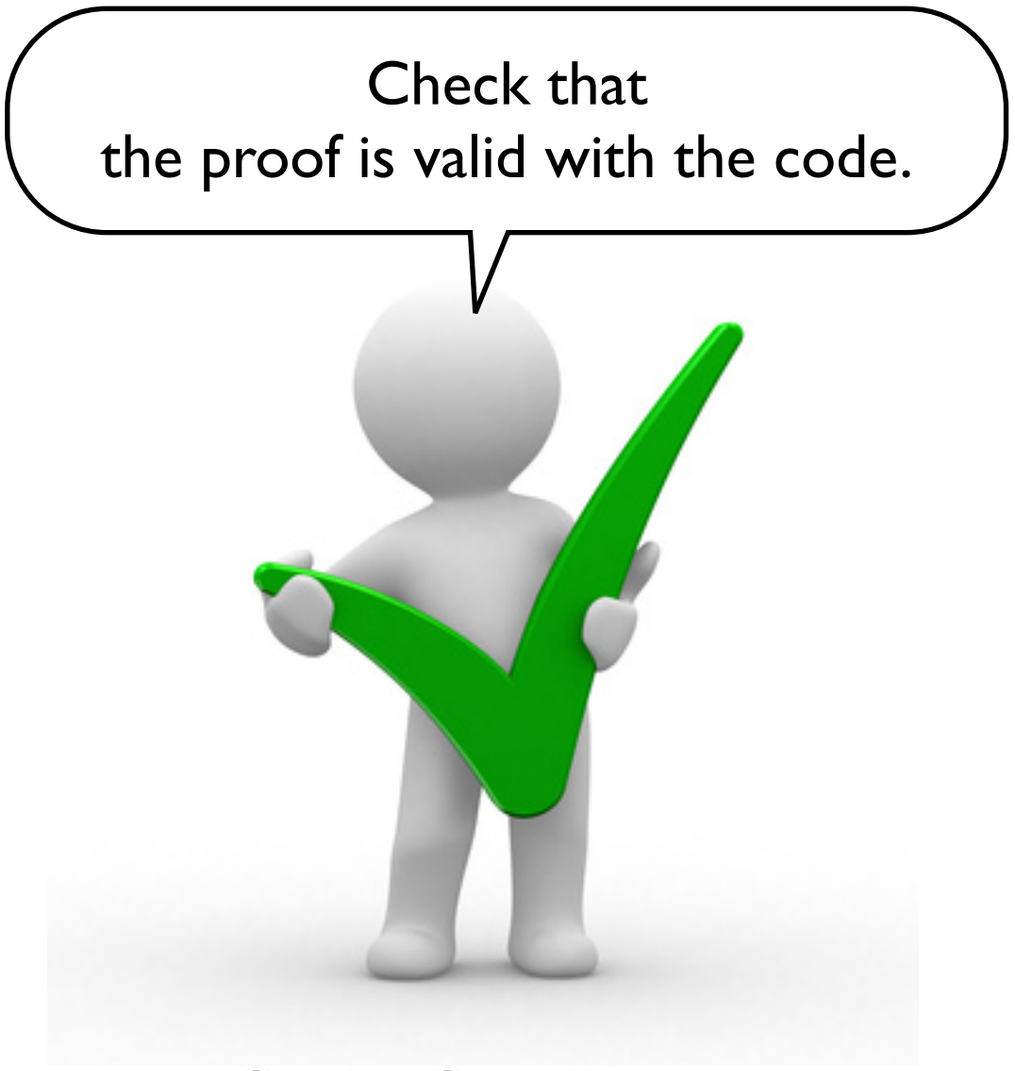
Issues in Proof-Carrying Code



Proof
(Certificate)

2. Size of the proof needs to be small enough.

Issues in Proof-Carrying Code



Code Consumer

3. Checking procedure needs to be done efficiently.

Issues in Proof-Carrying Code

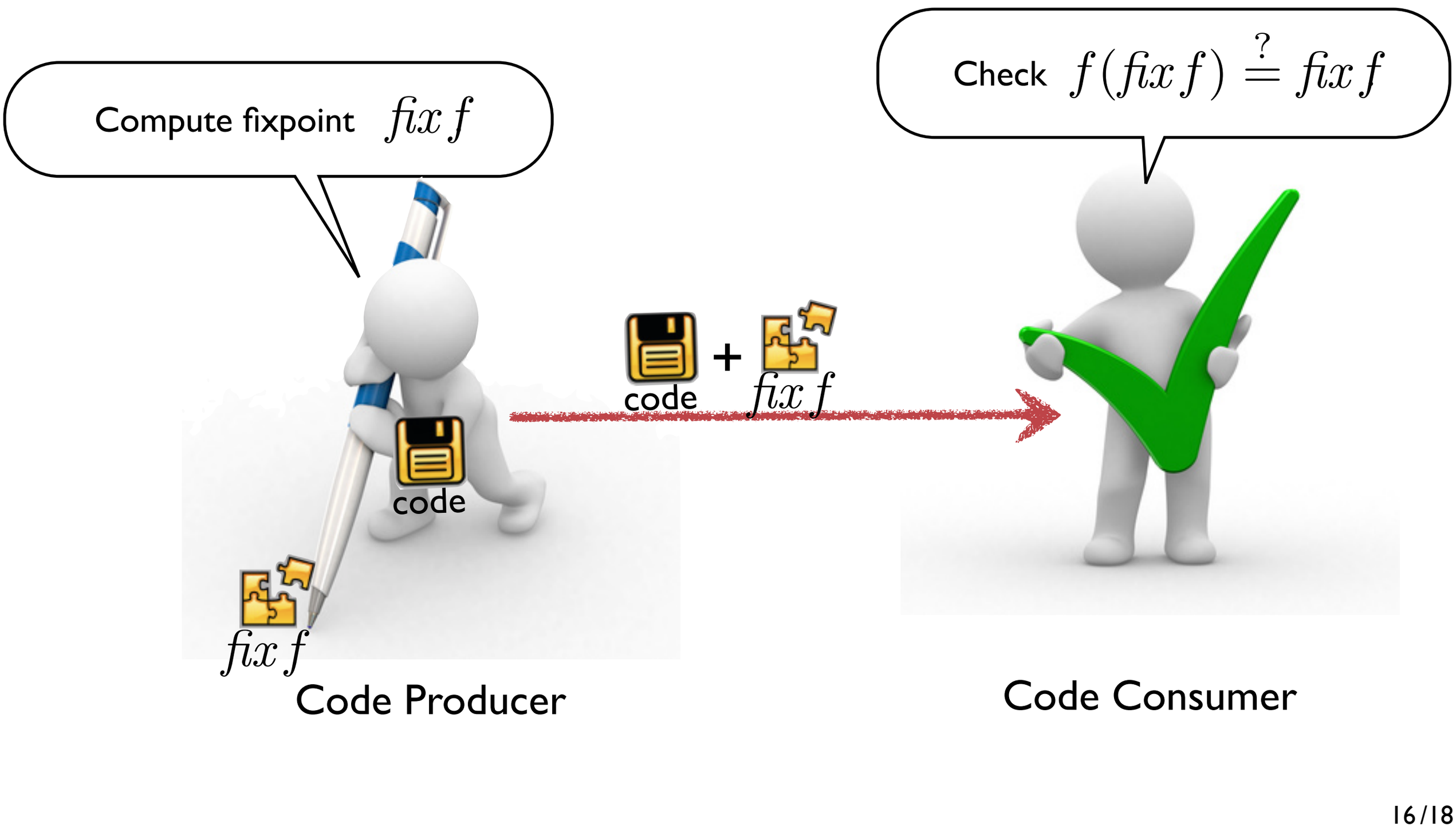
Checker

4. Checker needs to be simple and small enough.

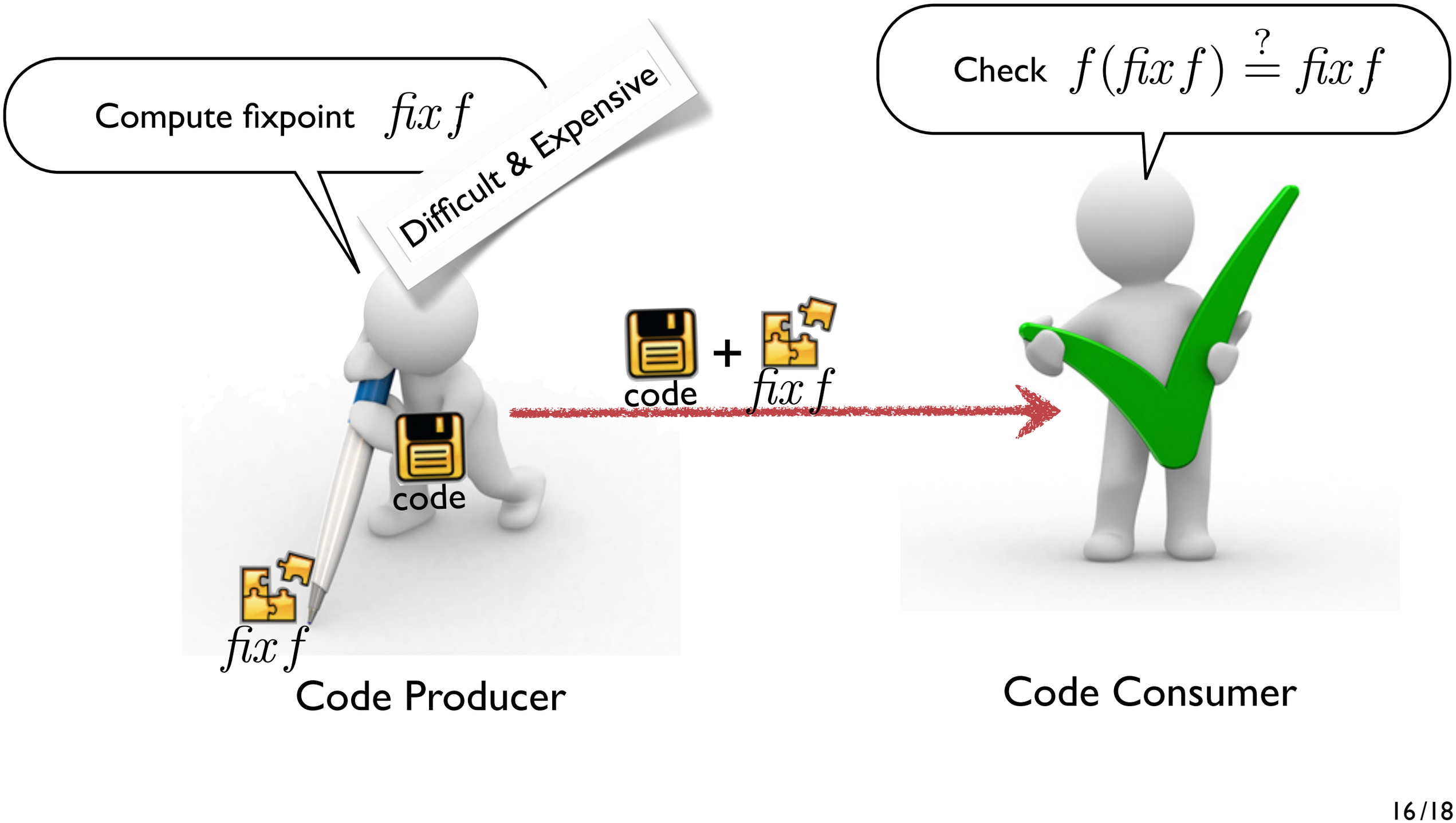
Abstraction-Carrying Code

- Derivation of proof-carrying code framework.
- “Fixpoint-Carrying Code”
  - Code producer computes fixpoint and send it with the code. (need full analyzer)
  - Code consumer checks that the received fixpoint is indeed a fixpoint for the received code.
    (simplified one-pass analyzer is enough)

Abstraction-Carrying Code



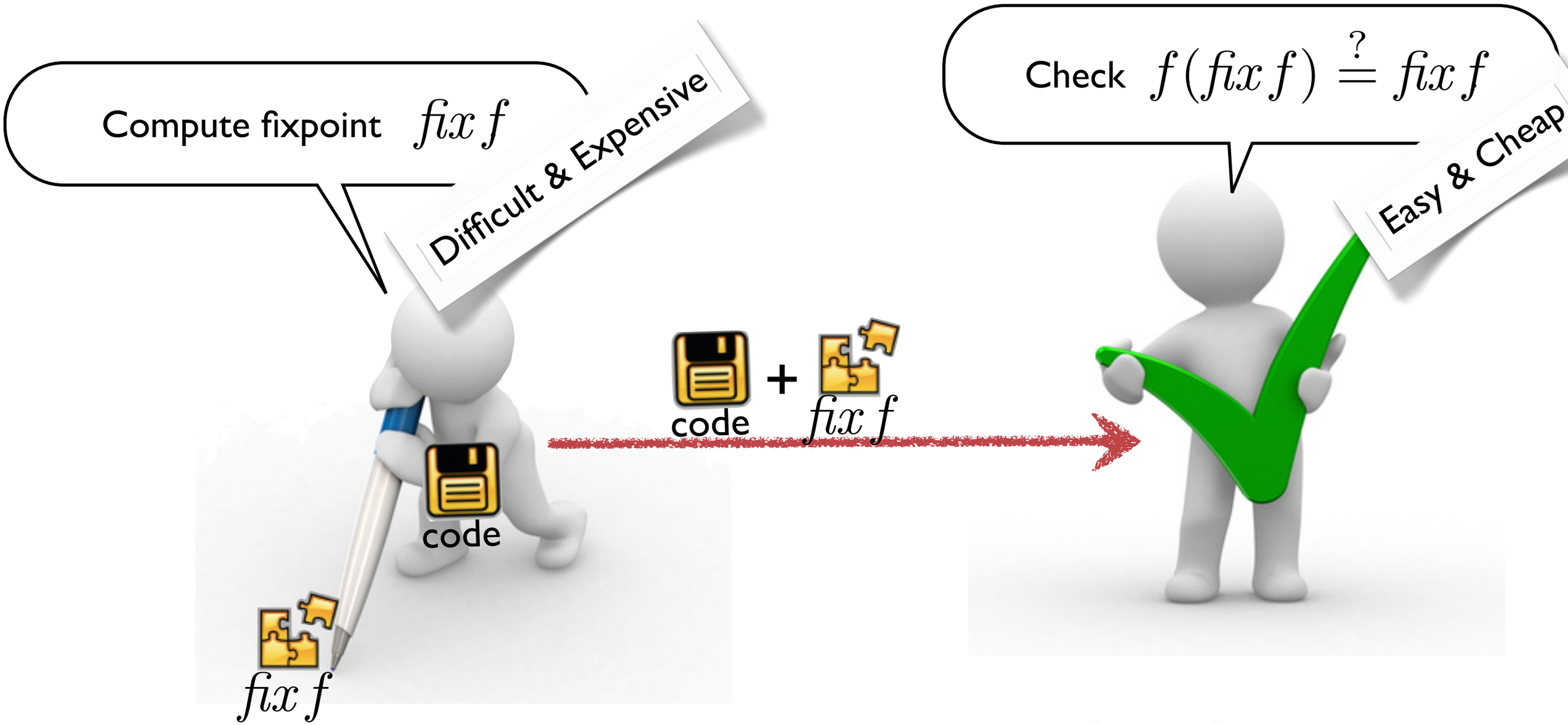
Abstraction-Carrying Code



Code Producer

Code Consumer

Abstraction-Carrying Code



Code Producer

Code Consumer

PPDP'05

Abstraction Carrying Code and Resource-Awareness

Manuel V. Hermenegildo[1,3] Elvira Albert[2] Pedro López-García[1] Germán Puebla[1]

[1] School of Computer Science
T. U. of Madrid (UPM)
Madrid, Spain
{herme,german,pedro.lopez}@fi.upm.es

[2] DSIP
Complutense U. of Madrid
Madrid, Spain
elvira@sip.ucm.es

[3] Depts. of CS and ECE
U. of New Mexico
Albuquerque, NM, USA
herme@unm.edu

ABSTRACT

Proof-Carrying Code (PCC) is a general approach to mobile code safety in which the code supplier augments the program with a certificate (or proof). The intended benefit is that the program consumer can locally validate the certificate w.r.t. the "untrusted" program by means of a certificate checker—a process which should be much simpler, efficient, and automatic than generating the original proof. *Abstraction Carrying Code* (ACC) is an enabling technology for PCC in which an *abstract model* of the program plays the role of certificate. The generation of the certificate, i.e., the abstraction, is automatically carried out by an abstract interpretation-based analysis engine, which is parametric w.r.t. different abstract domains. While the analyzer on the producer side typically has to compute a semantic fixpoint in a complex, iterative process, on the receiver it is only necessary to check that the certificate is indeed a fixpoint of the abstract semantics equations representing the program. This is done in a single pass in a much more efficient process. ACC addresses the fundamental issues in PCC and opens the door to the applicability of the large body of frameworks and domains based on abstract interpretation as enabling technology for PCC. We present an overview of ACC and we describe in a tutorial fashion an application to the problem of resource-aware security in mobile code. Essentially the information computed by a cost analyzer is used to generate *cost certificates* which attest a safe and efficient use of a mobile code. A receiving side can then reject code which brings cost certificates (which it cannot validate or) which have too large cost requirements in terms of computing resources (in time and/or space) and accept mobile code which meets the established requirements.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers, Validation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Diagnostics, Symbolic execution*; F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.3.2 [**Programming Languages**]: Language Classifications—*Constraint and logic languages, Multiparadigm languages*.

General Terms

Reliability, Security, Languages, Theory, Verification.

Keywords

Program verification, mobile code certification, resource awareness, program debugging, cost analysis, granularity control, distributed programming, abstract interpretation, programming languages.

PPDP'05, July 11–13, 2005, Lisbon, Portugal.

1. INTRODUCTION

In traditional distributed execution receivers are assumed to be either dedicated and/or to trust and simply accept (or take, in the case of work-stealing schedulers) available tasks. Typically, tasks are run at the receiving end under some administrative domain that is previously agreed on by producer and consumer of the task. However, many recently proposed applications (such as peer-to-peer systems, the GRID, and other similar overlay computing systems) represent more open settings where the administrative domain of the receiver can be completely different from that of the producer. Also, in these applications the receiver is typically being used for other purposes (e.g., as a general-purpose workstation) in addition to being a party to the distributed computation.

In such an environment, interesting security- and resource-related issues arise. In particular, in order to accept some code and a particular task to be performed, the receiver must clearly have some assurance of the *correctness and characteristics of the code received* and also of *the kind of load the particular task is going to pose.* A receiver should be free to reject code that does not adhere to a particular *safety policy* involving both more traditional safety issues (such as, e.g., that it will not write on specific areas of the disk) and *resource*-related issues (such as, e.g., that it will not compute for more than a given amount of time, or that it will not take

Thank you