Techniques for Trusted Software Engineering

Premkumar Devanbu

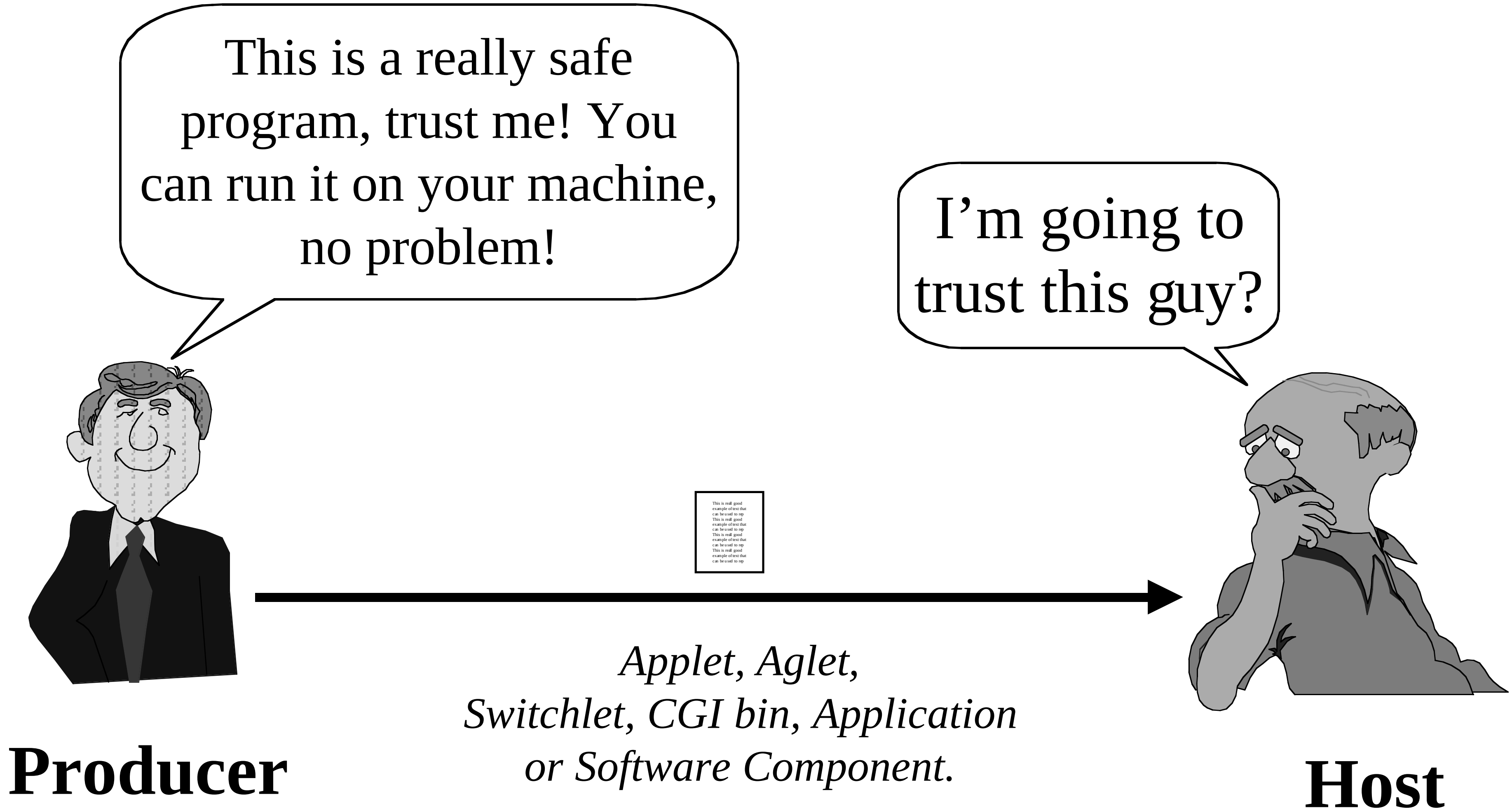
University of California, Davis

devanbu@cs.ucdavis.edu

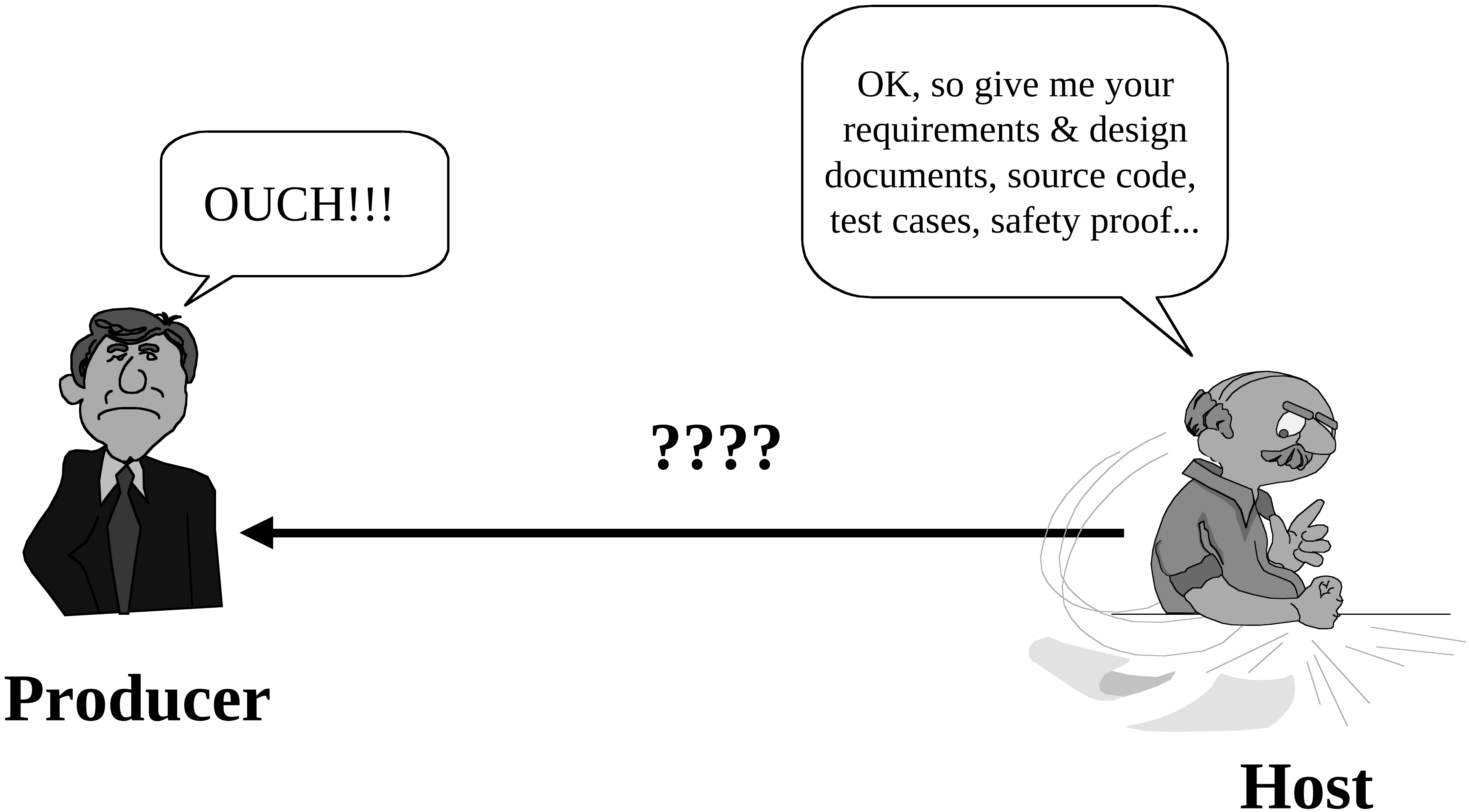
http://www.cs.ucdavis.edu/~devanbu

(Collaborators:
Jonathan Peters, Henry Naftulin UC Davis
Dr. S. Stubblebine, AT&T Research,
P.W. Fong, Simon Fraser University)

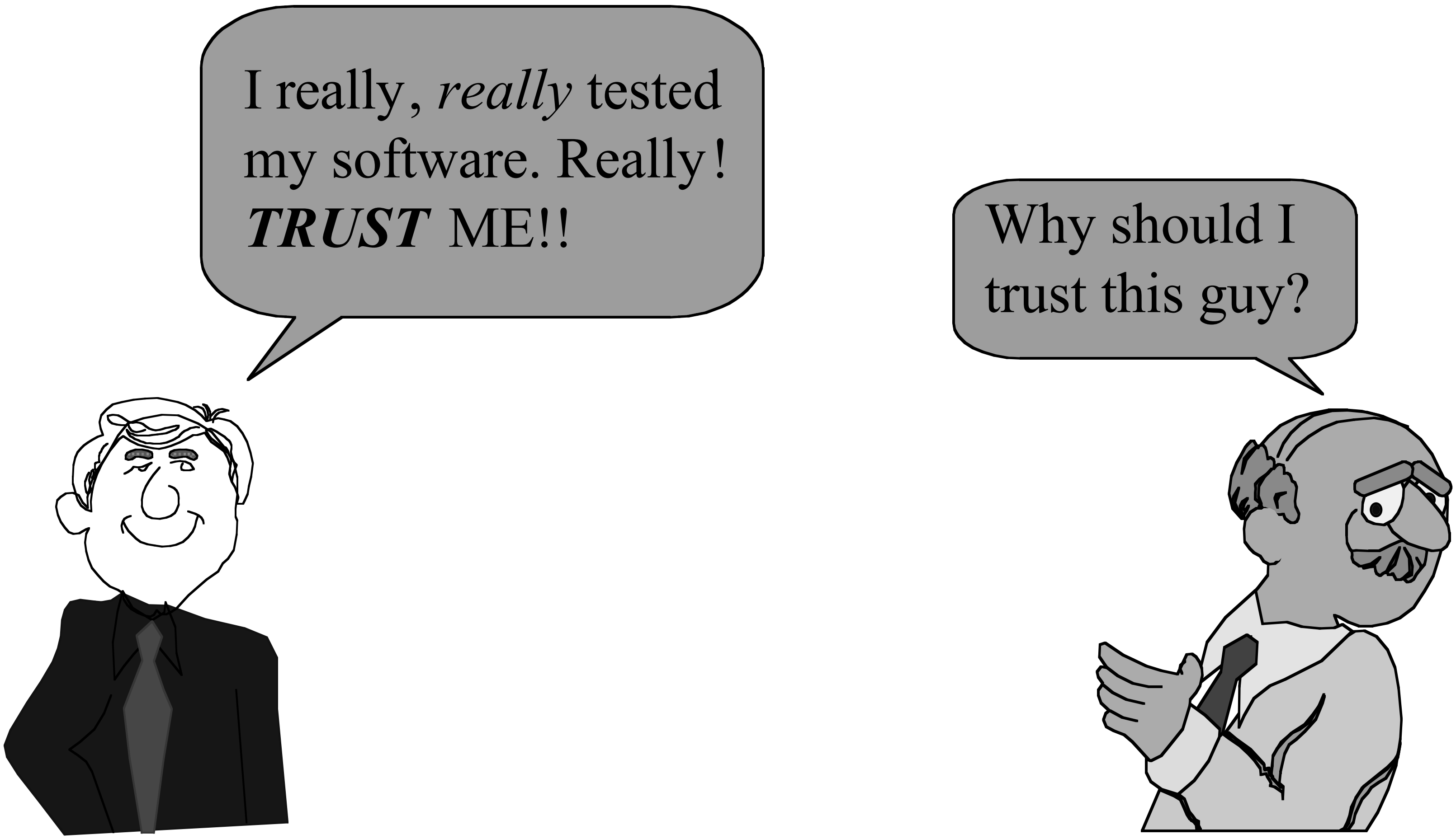
The Problem.


This is a really safe program, trust me! You can run it on your machine, no problem!
I'm going to trust this guy?
Applet, Aglet, Switchlet, CGI bin, Application or Software Component.
Producer
Host

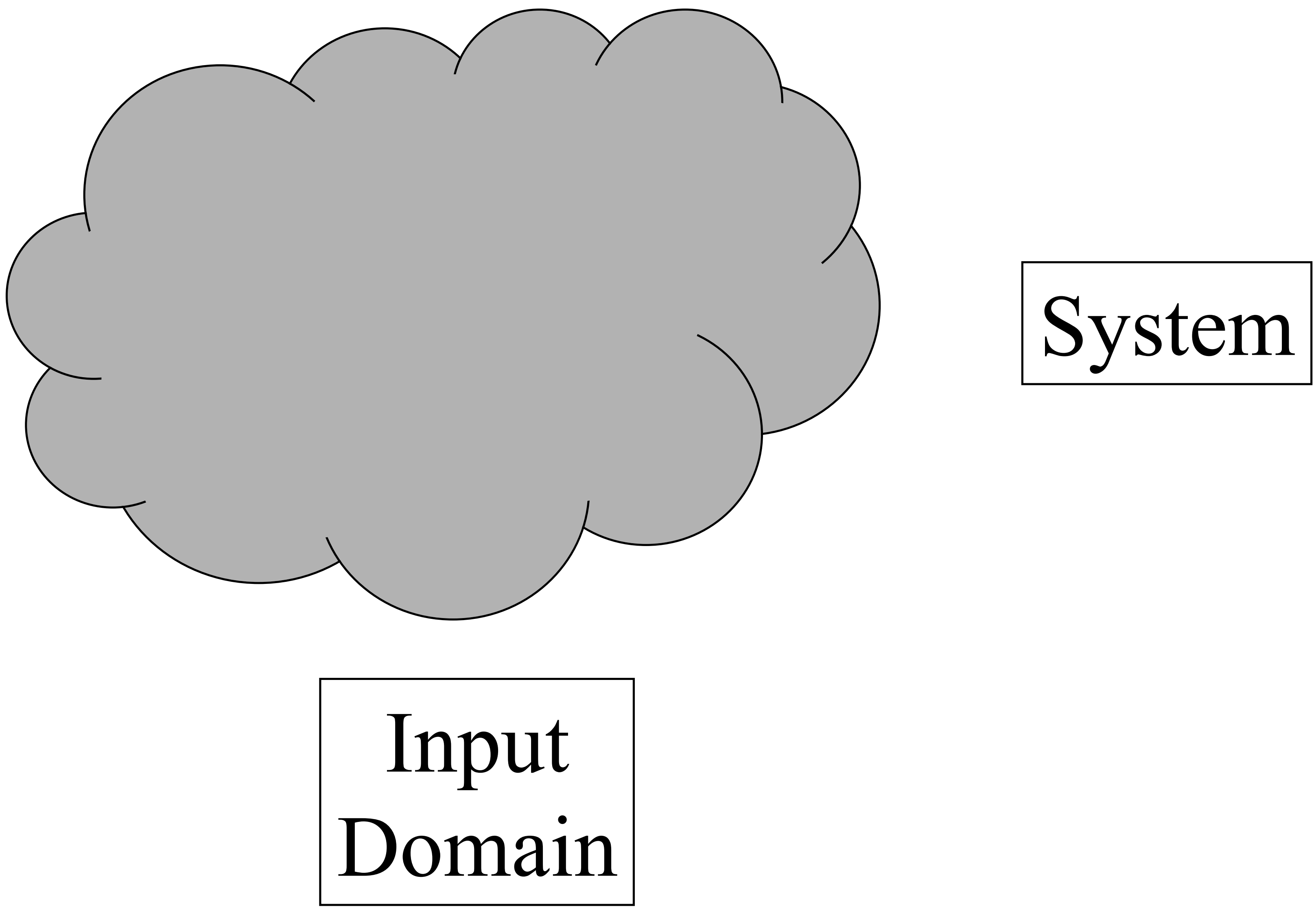
One partial solution

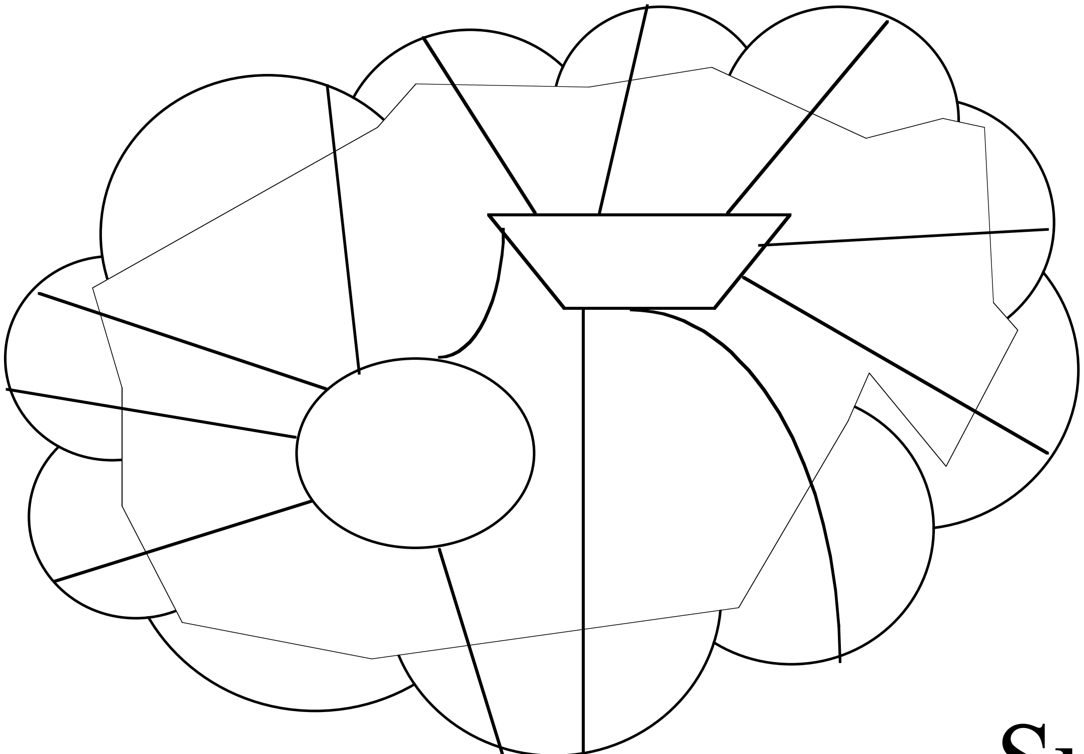


Verifying testing processes



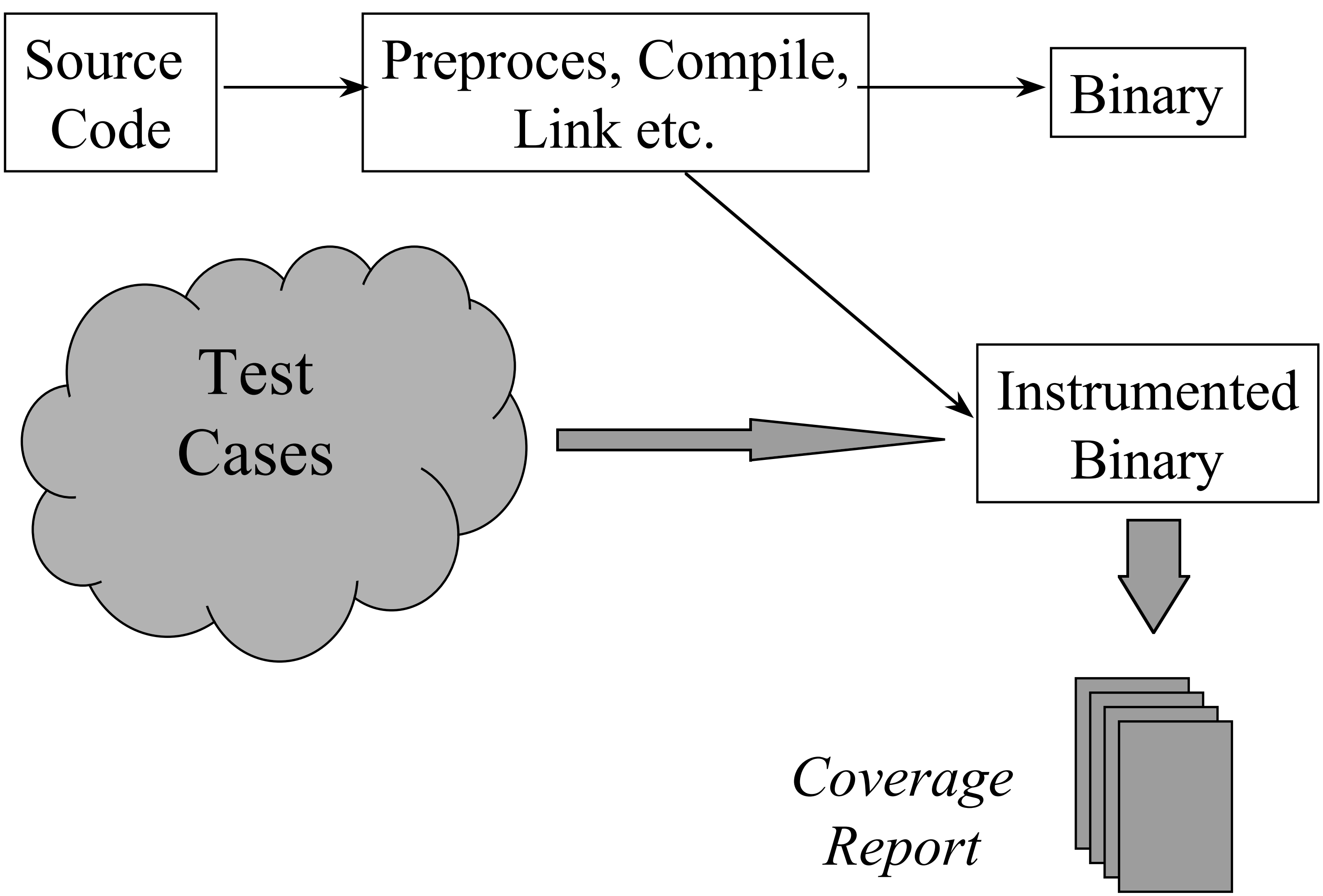
When is testing complete?



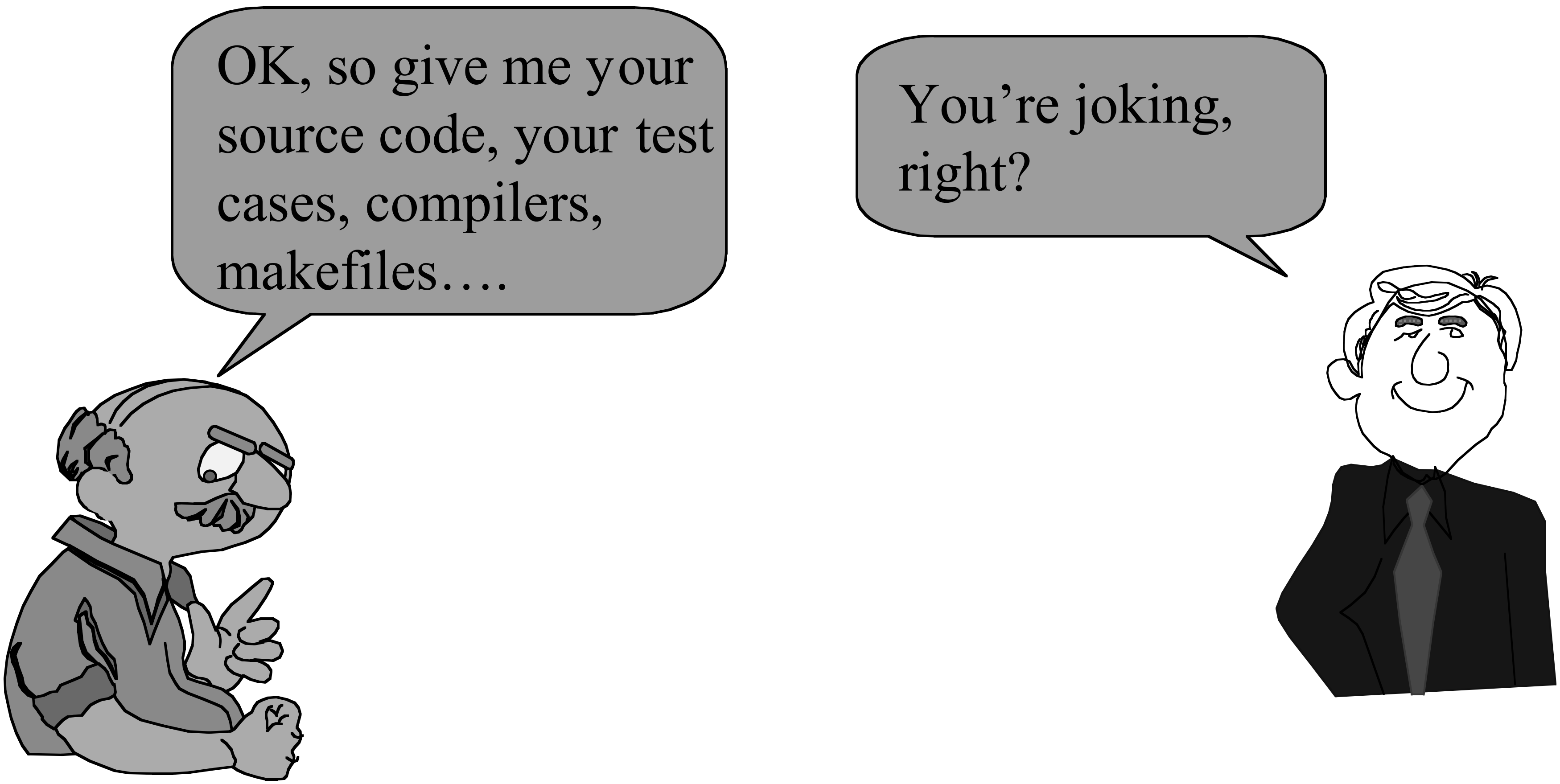


Sub-domain testing

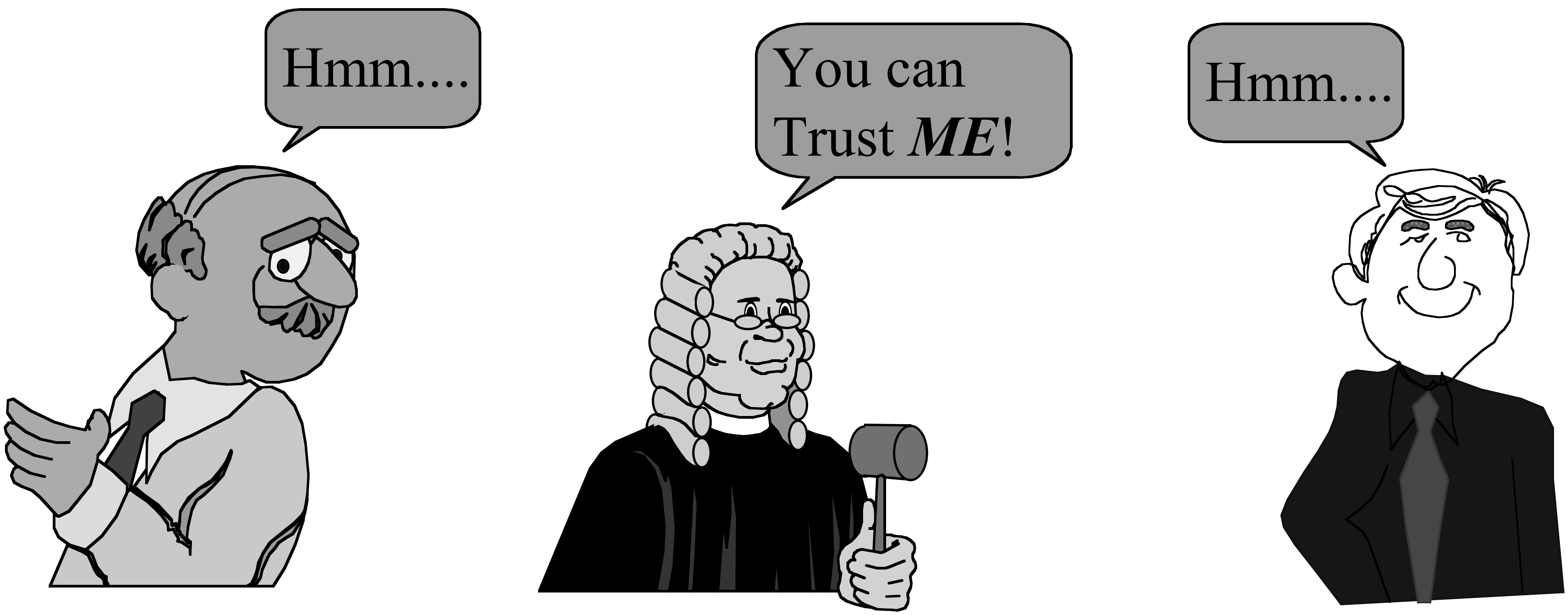
Coverage Measurement



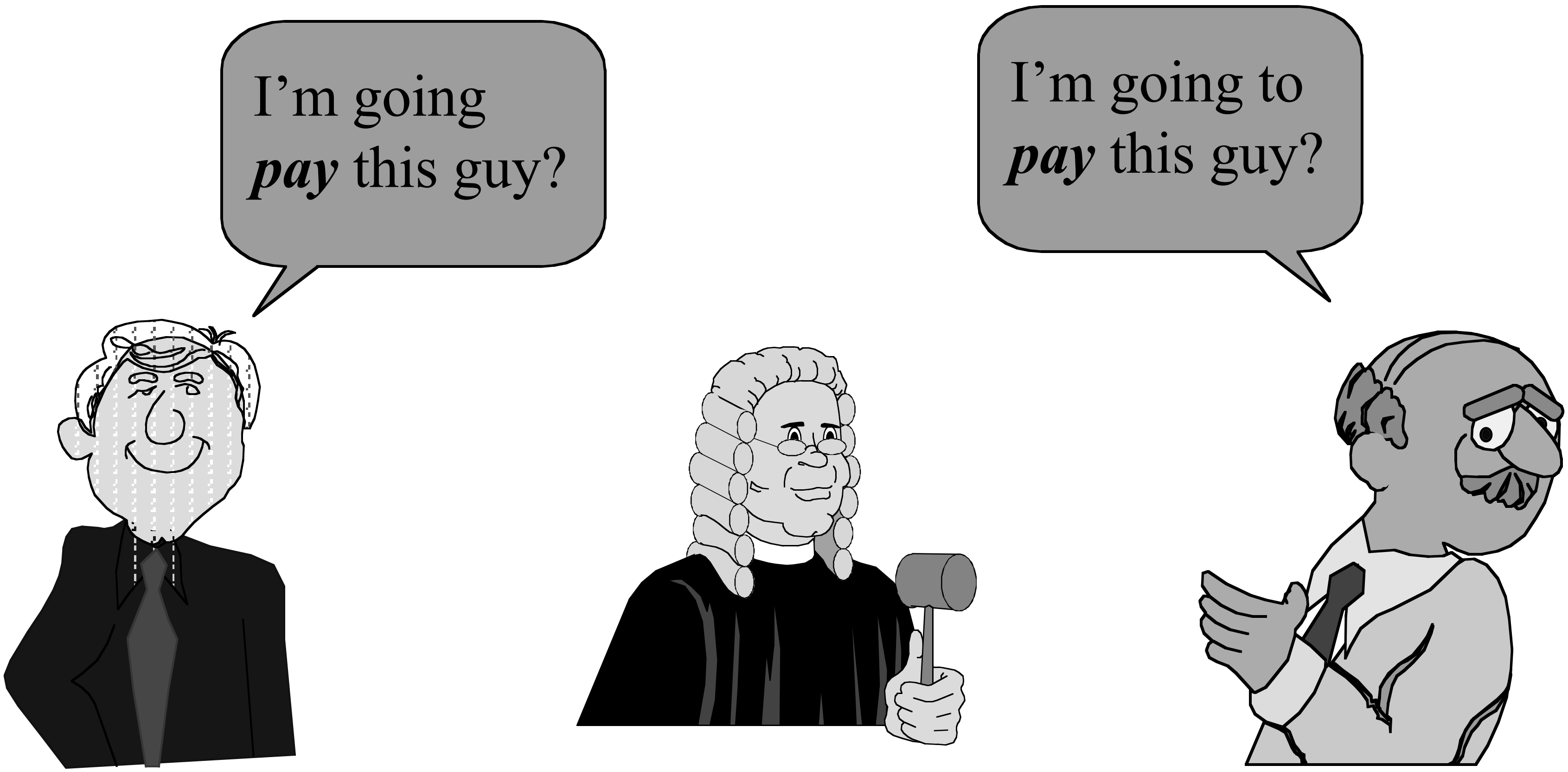
Verifying testing processes.



Trusted Third Party



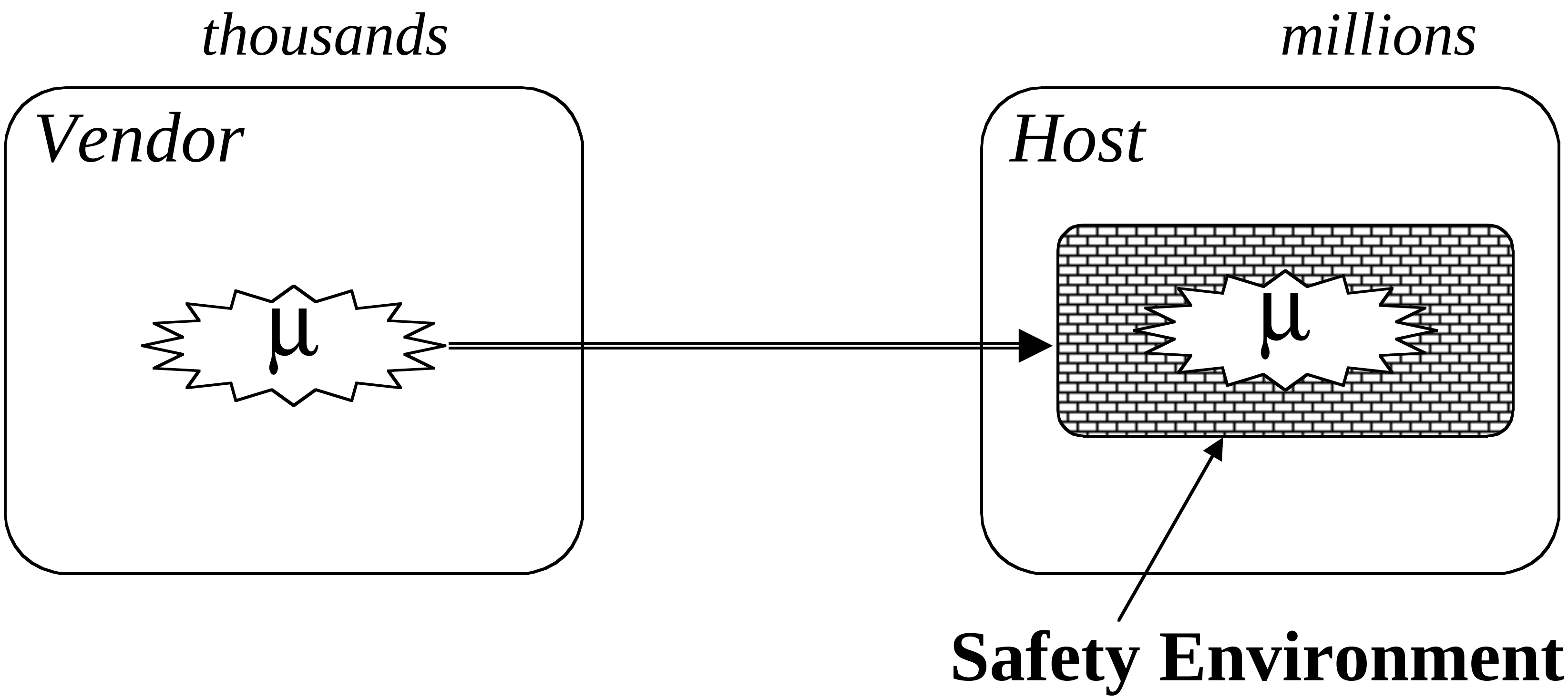
Problem 1 with Basic 3rd P. M.



Results and Publications

- What were the ***software processes*** that the producer used? Specification, Design, Testing?
  (Devanbu & Stubblebine, FSE ‘97, Zurich)
- What can I find from/about from the ***software product*** itself, and what can the producer tell me about it?
  (Devanbu & Stubblebine, ASE ‘97
  Devanbu, Fong & Stubblebine ICSE ‘98
  Devanbu & Stubblebine, Oakland ‘98)

Mobile Code (Java, ActiveX,…)

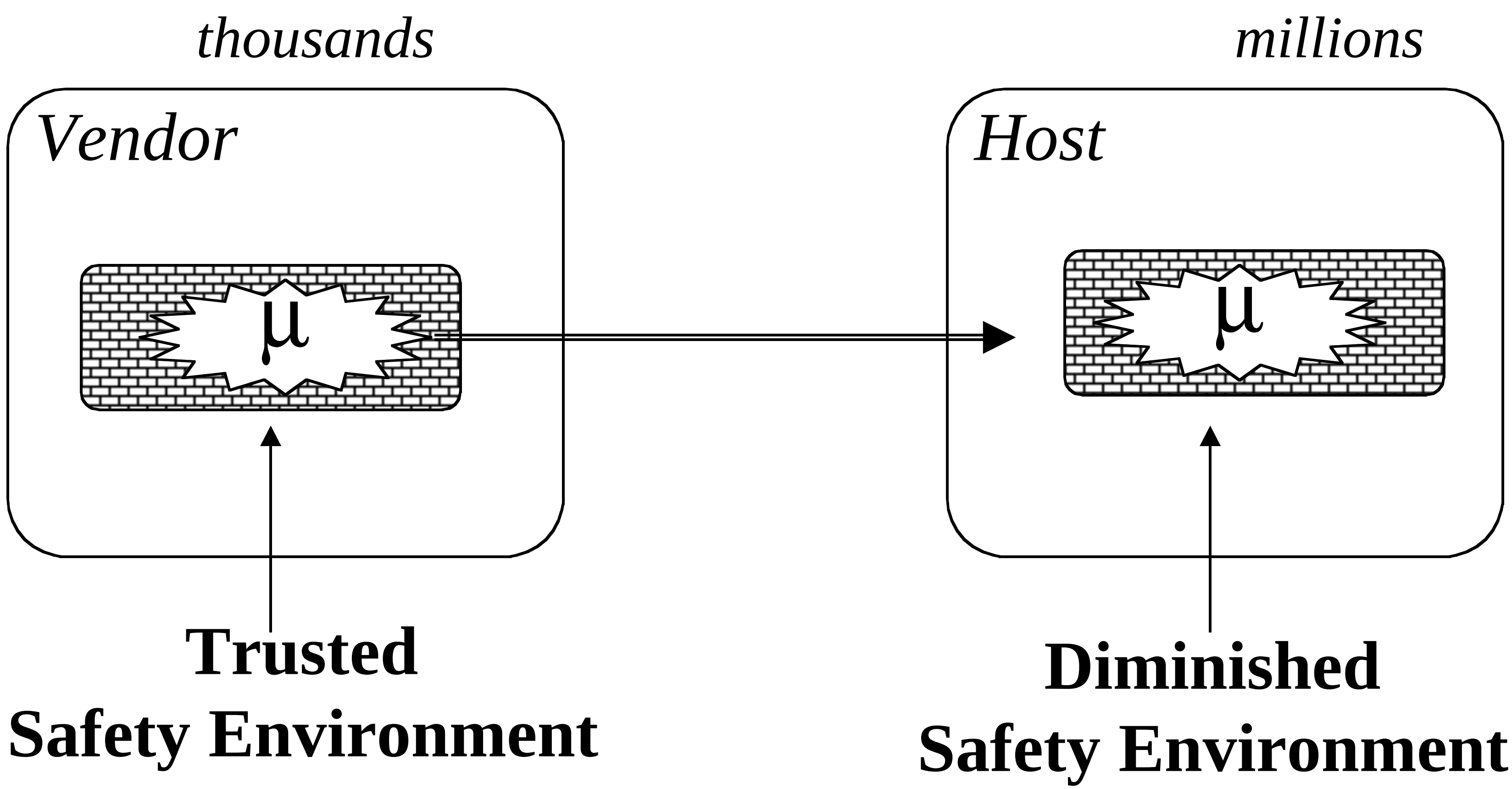


Dimensions of Trust in Mobile Code

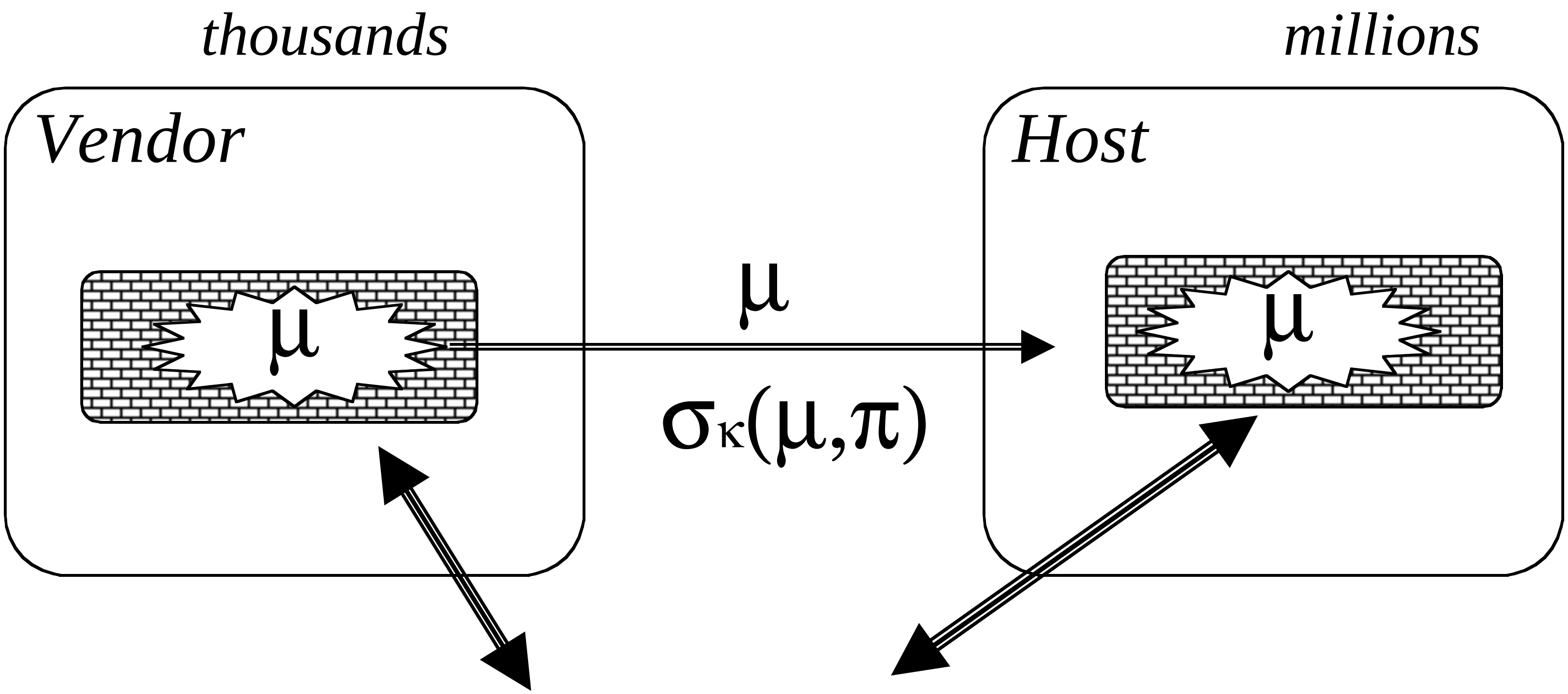
- *Safety:* What properties does the host care about?
- *Efficiency:* What's the runtime impact?
- *Cost:* Additional time, personnel, complexity etc?
- **** Disclosure**:* What does the producer have to reveal?
- **** Configuration**:* how hard is to upgrade the host's checking mechanism when a weakness is discovered?

** Software Engineering Issues!*

Our Proposal -- Part 1



Our Proposal--Part 2



Trusted Configuartion Management

Java

- *Safety Property:* Bytecode is well-typed.
- *Disclosure:* Source code of the program.
- *Efficiency:* Dataflow-based typechecking (+ typesafe linking + runtime sandboxing).
- *Cost:* Minimal impact.
- *Configuration:* O(10^7) upgrades, when a flaw is found.

ActiveX

- *Safety Property:* None, based on signing.
- *Disclosure:* Nil (just binary)
- *Efficiency:* High (signature checking + no runtime load).
- *Cost:* Substantial for non-trusted parties.
- *Configuration:* None, unless fundamentally changed.

Proof Carrying Code

(http://www.cs.cmu.edu/~necula)

- *Safety Property:* Specified in formal logic by host.
- *Disclosure:* Invariant assertions + proof.
- *Cost*: Producer creates *complete, formal proof of safety!!!*
- *Efficiency:* low: ~100's ms for small programs (100's bytes)
- *Configuration:* One per host, when faults in proof checker are discovered.

Another Approach

- Producer creates software.
- The binary (or bytecode) is submitted to a *trusted tool encased in a trusted computer*.
- The trusted tool verifies that the software has the desired property, and appends a statement, and a signature.
- Host *just checks the signature*, and *runs*!
- Key management for configuration management.

Java (re-visited)

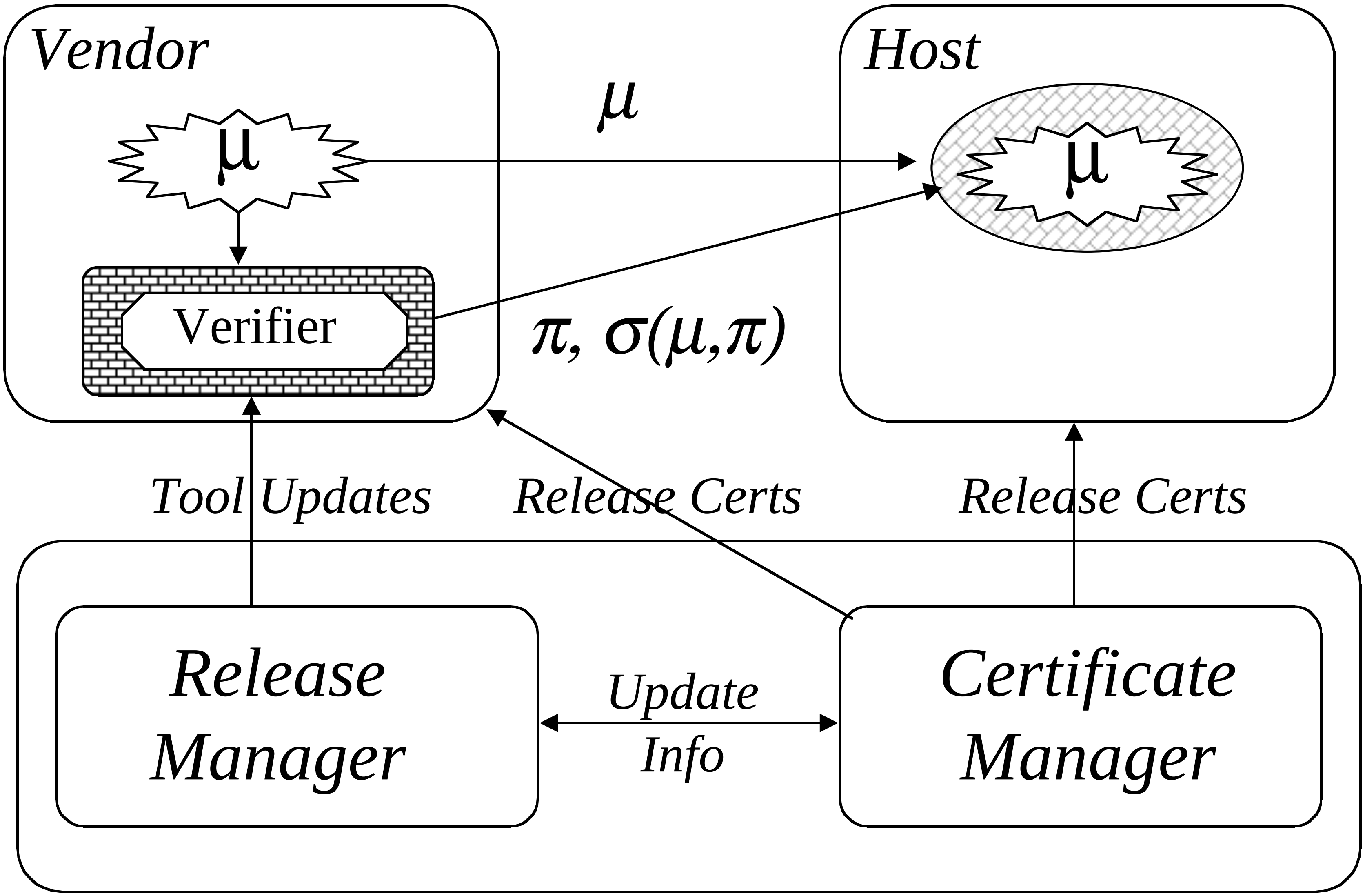
- *Safety Property:* Bytecode verified *at producer's site.*
- *Disclosure:* Source code of the program.
- *Efficiency:* signature checking (+ typesafe linking + runtime sandboxing).
- *Cost:* low to moderate.
- *Configuration:* Just key mgmt for hosts, software updates for producers.

Proof-Carrying Code (re-visited)

Producer creates binary, assertions, and proofs. Annotated binary checked at producer's site, *unannotated* binary signed & delivered.

- *Safety Property:* Host's choice.
- *Disclosure:* ***Nothing beyond binary!!***
- *Efficiency:* Very high. Signature check only.
- *Upgrades:* Key management + checking software upgrades.

Key management for Configuration

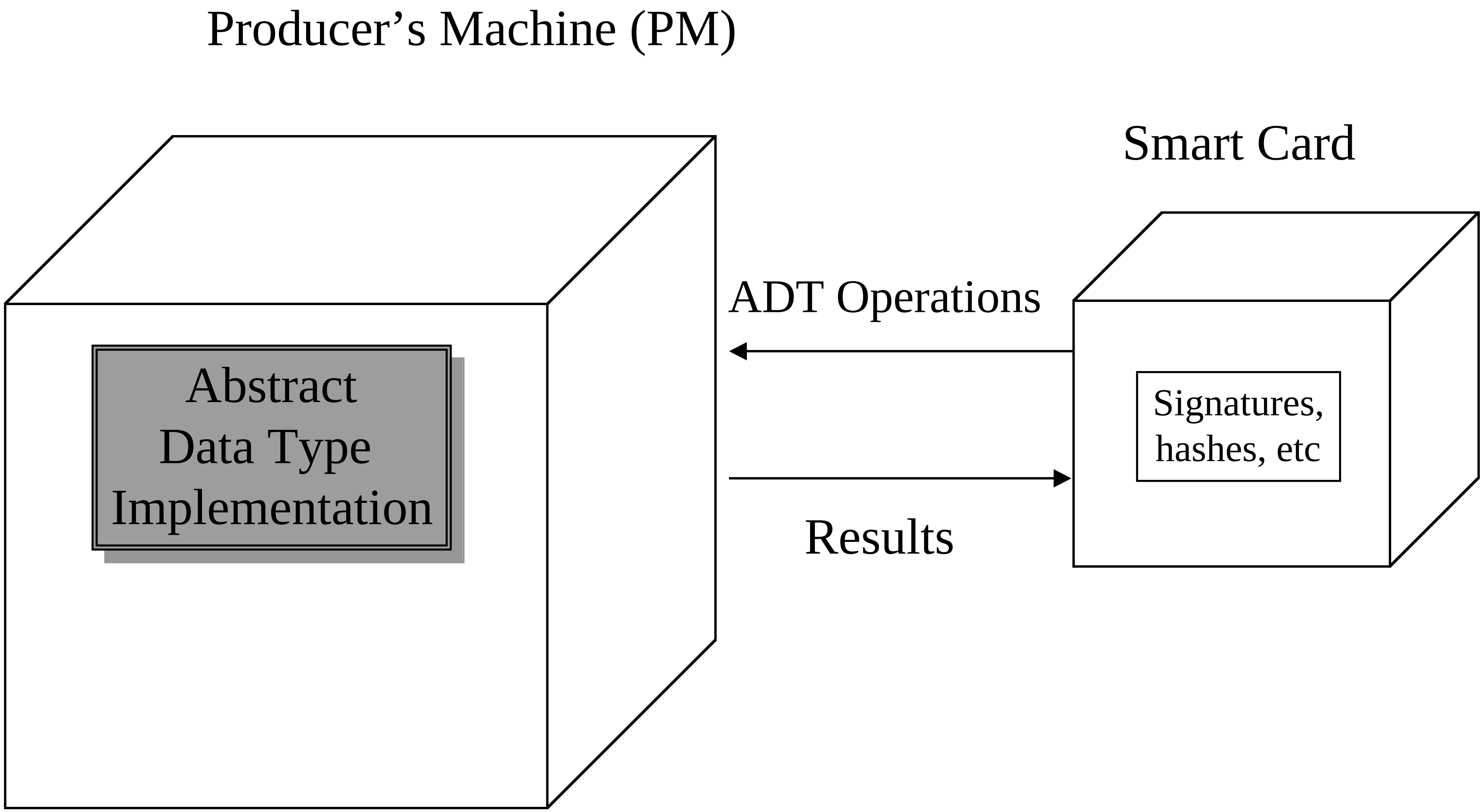


Difficulty: Resource limitation in smart cards: Form factor + physical security + heat dissipation constraints.

Can we use “host” computer, at producer’s site, to off-load some computation?

But, the Producer’s Machine is hostile!!!

Idea: Leverage <u>limited</u> resources in smart card to enforce integrity of much more resource-intensive computations.


Producer's Machine (PM)
Abstract
Data Type
Implementation
ADT Operations
Results
Smart Card
Signatures,
hashes, etc

New:

$\underline{\sigma = \textit{Random}()}$

Card (to Producer's Machine PM): New Stack(σ)

Push(item):

Card(to PM): item, σ

$\underline{\sigma = \textit{Signature}(\textit{Append}(\text{item}, \sigma))}$

Pop:

Card(to PM): Do a Pop!

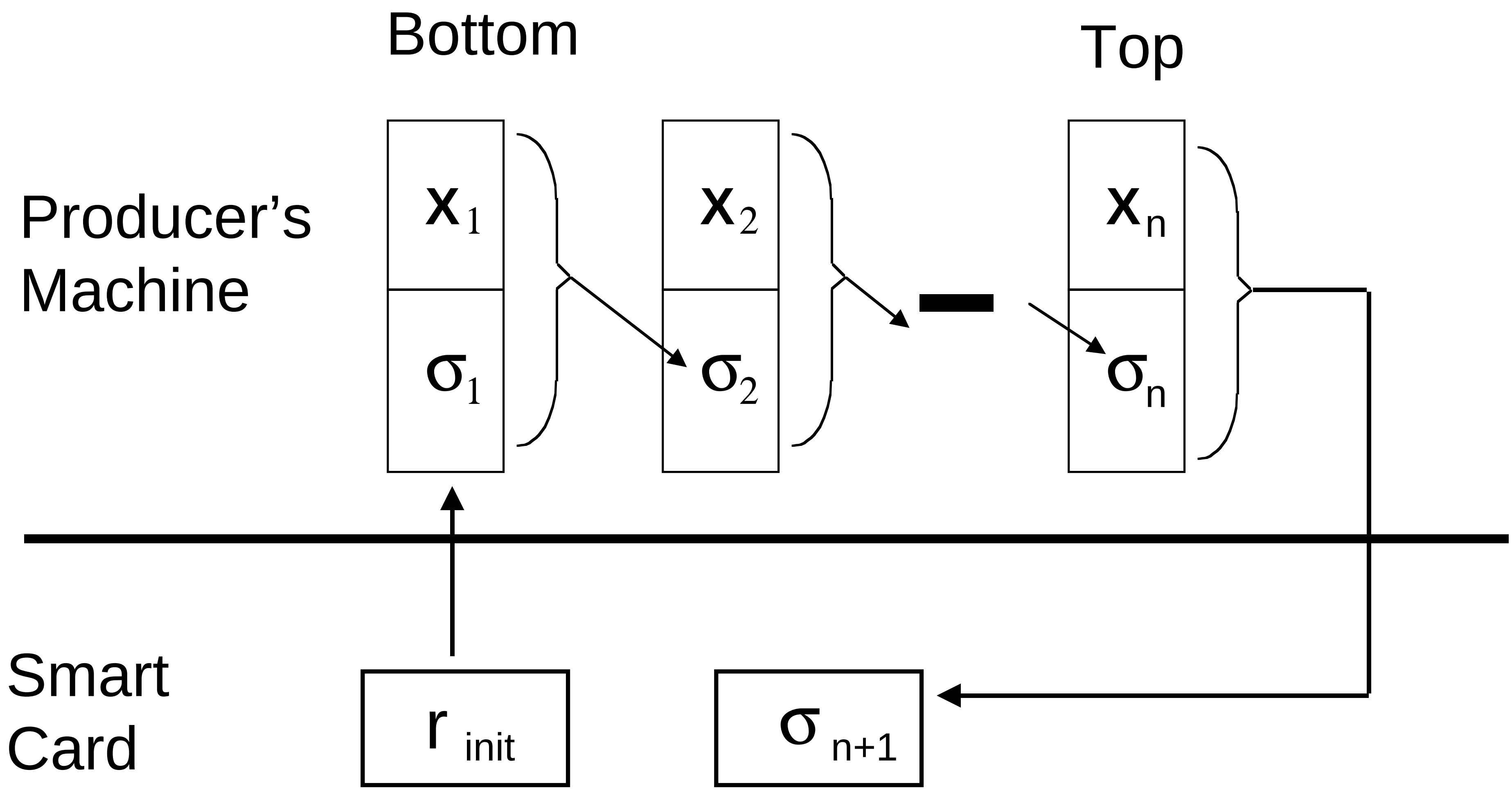
PM(to Card): item, oldσ

Is $\underline{\sigma = \textit{Signature}(\textit{Append}(\text{item}, \text{old}\sigma))}$??

If yes, $\underline{\sigma = \text{old}\sigma}$ and continue...

If not, terminate!

O(1) bits in smart card, O(1) bits transferred per operation, O(1) signature computations per operation.



Bottom
Top
Producer's Machine
x_1
σ_1
x_2
σ_2
x_n
σ_n
Smart Card
r_{init}
σ_{n+1}

Problem: Trusted software engineering.

Approaches:

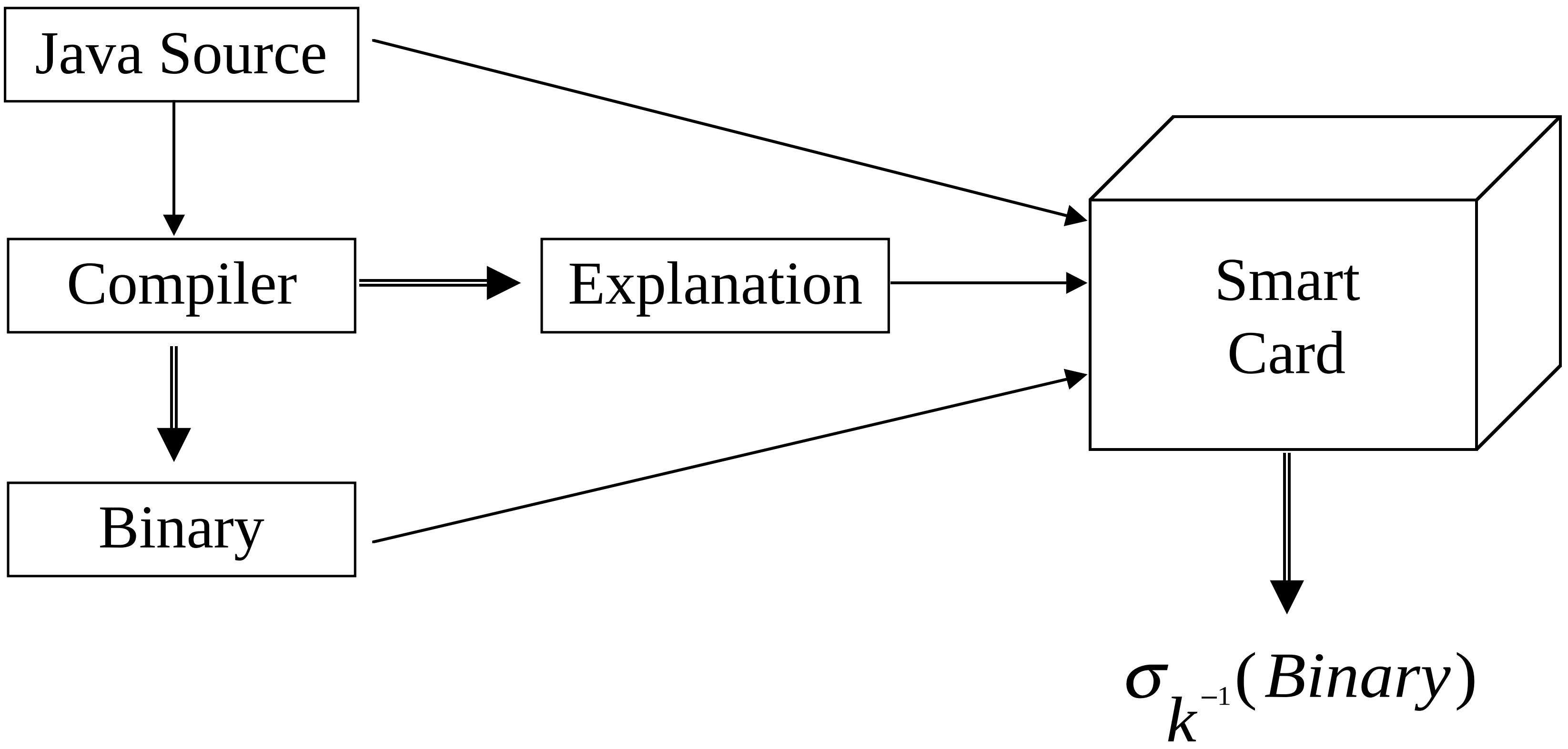
Trust in Process: Cryptographic test coverage verification, ACM SIGSOFT FSE 96.

Trust in Product: Software tools embedded in physically secure processors, with associated key management infrastructure.

Conclusions

- Two issues with current approach to trusted software engineering: *configuration management*, and *disclosure of intellectual property*.
- Our approach: Using a trusted tool at the code producer's site, along with trusted configuration management.
- Interesting software engineering challenge: resource limitation on trusted computers.

Compiled Java Scenario



Building in the Policy