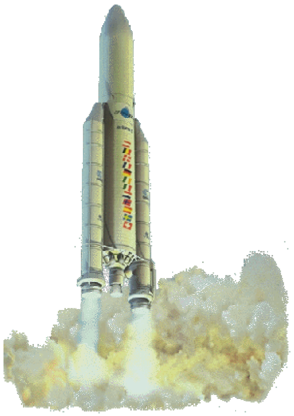
Lectures on Proof-Carrying Code

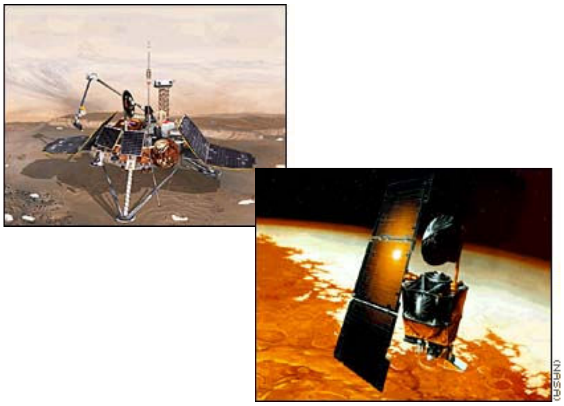
Peter Lee

Carnegie Mellon University

Lecture 1 (of 3)
June 21-22, 2003
University of Oregon

2004 Summer School on Software Security



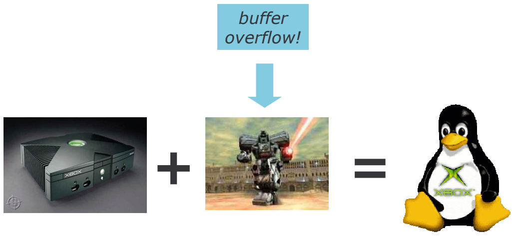


(NASA)

"After a crew member mistakenly entered a zero into the data field of an application, the computer system proceeded to divide another quantity by that zero. ***The operation caused a buffer overflow, in which data leaked from a temporary storage space in memory, and the error eventually brought down the ship's propulsion system.*** The result: ***the USS Yorktown was dead in the water for more than two hours***."

According to CERT, buffer overflow attacks are the #1 exploit for network security attacks.

http://www.cert.org/summaries/



Automotive analogy

"If the automobile had followed the same development as the computer, a Rolls-Royce would today cost $100, get a million miles per gallon, and ...

Automotive analogy

"If the automobile had followed the same development as the computer, a Rolls-Royce would today cost $100, get a million miles per gallon, and explode once a year killing everyone inside."

- Robert Cringely

Cars in the Real World

Problems at Mercedes:

- Forced to buy back 2000 copies of the latest E-Class sedan, due to problems with computing and telecommunications systems
- J.D.Power initial quality rankings have dropped to 15th (even below Chevrolet!) in 2003
  - board member Jurgen Hubbert says this is directly related to the effort to be a "technology leader"

Observations

Many failures are due to simple problems "in the details"

Code reuse is necessary but perilous

Updateable/mobile code is essential

Performance matters a lot

Opportunities

Progress depends fundamentally on our ability to *reason about programs.*

The opportunities are great.

Who will provide the technology for systems that work?

About these lectures

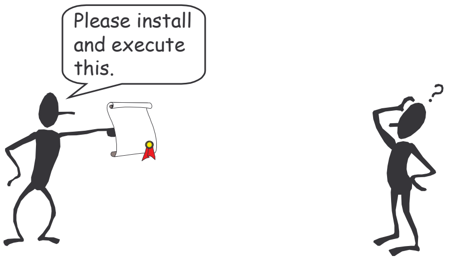
The main topic is *proof-carrying code*, an example of *certified code*

It won't be possible to go through all aspects in complete detail

But I hope to provide background to make it easier to get started with research

The Code Safety Problem

The Code Safety Problem

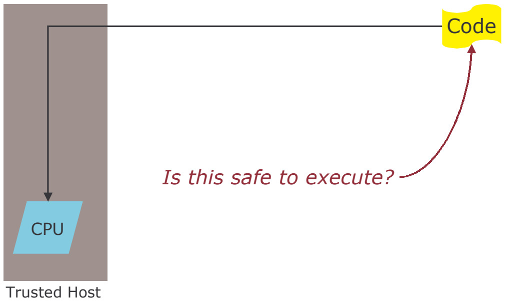


"Applets, Not Craplets"

- Luca Cardelli, 1996

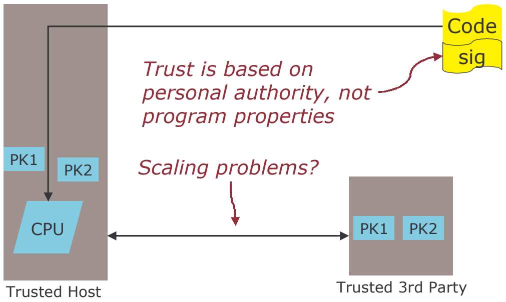
> “If you have ‘process’
> without ‘inspiration,’ all you
> end up with is well-documented crap.”
>
> — *Dr. John C. Sommerer,*
> *CTO, Johns Hopkins*
> *Advanced Physics Lab*

Code Safety



Is this safe to execute?

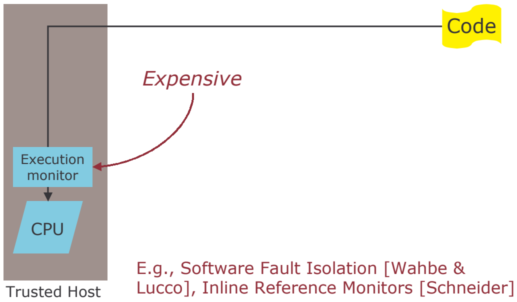
Approach 1
Trust the code producer



Trust is based on personal authority, not program properties

Scaling problems?

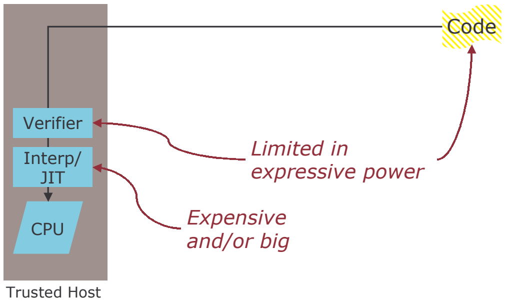
Approach 2
Baby-sit the program



Expensive

E.g., Software Fault Isolation [Wahbe & Lucco], Inline Reference Monitors [Schneider]

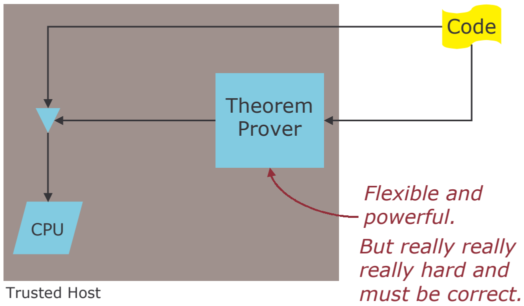
Approach 3
Java



Limited in expressive power

Expensive and/or big

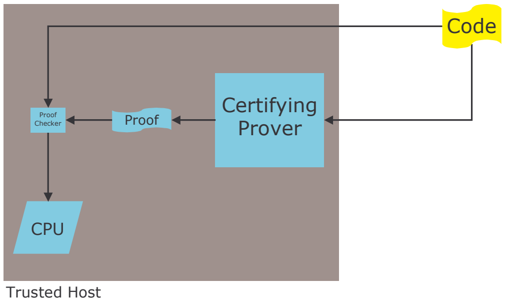
Approach 4
Formal verification



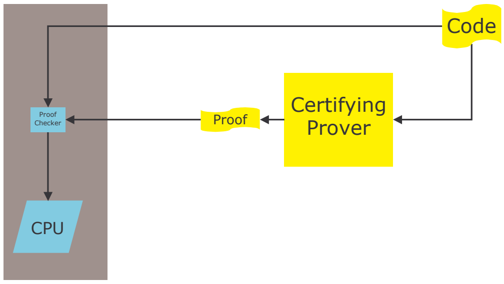
Flexible and powerful.

But really really really hard and must be correct.

A key idea: Checkable certificates

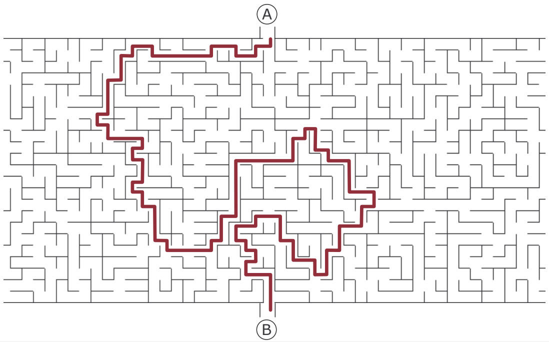


A key idea: Checkable certificates



Proof-Carrying Code

[Necula & Lee, OSDI'96]



Five Frequently Asked Questions

Question 1

How are the proofs represented and checked?

Formal proofs

Write "x is a proof of predicate P" as **`x:P`**.

What do proofs look like?

Example inference rule

If we have a proof x of P and a proof y of Q, then x and y together constitute a proof of P ∧ Q.

$$\frac{\Gamma \vdash x : P \qquad \Gamma \vdash x : Q}{\Gamma \vdash (x, y) : P \wedge Q}$$

Or, in ASCII:

- **`Given x:P, y:Q then (x,y):P*Q.`**

More inference rules

Assume we have a proof x of P. If we can then obtain a proof b of Q, then we have a proof of P ⇒ Q.

- Given **`[x:P] b:Q then`**
  `fn (x:P) => b : P → Q.`

More rules:

- **`Given x:P*Q then fst(x):P`**
- **`Given y:P*Q then snd(y):Q`**

Types and proofs

So, for example:

```
fn (x:P*Q) => (snd(x), fst(x))
      : P*Q → Q*P
```

This is an ML program!

Also, typechecking provides a “smart” blackboard!

Curry-Howard Isomorphism

In a logical framework language, predicates can be represented as types and proofs as programs (i.e., expression terms).

Furthermore, under certain conditions ***typechecking is sufficient to ensure the validity of the proofs***.

“Proofs as Programs”

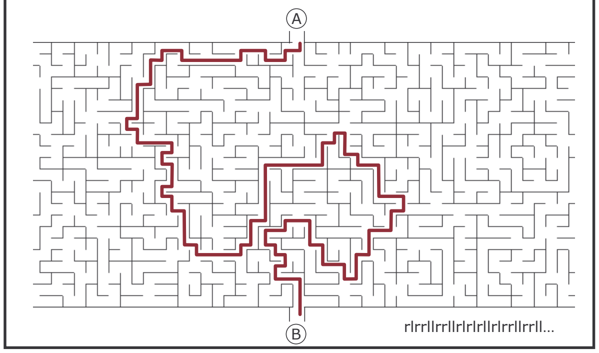
“Propositions as Types”

LF

The Edinburgh Logical Framework language, or LF, provides an expressive language for proofs-as-programs.

Furthermore, its use of dependent types allows, among other things, the axioms and rules of inference to be specified as well

Oracle strings

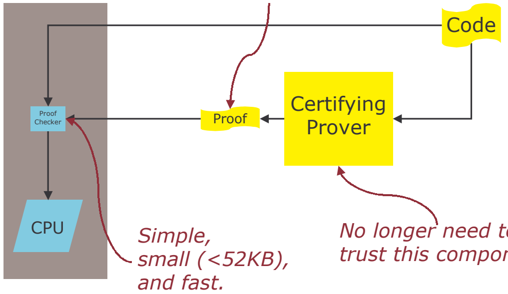


rlrrllrrllrlrlrllrlrrllrrll...

Question 2

How well does this work in practice?

The Necula-Lee experiments

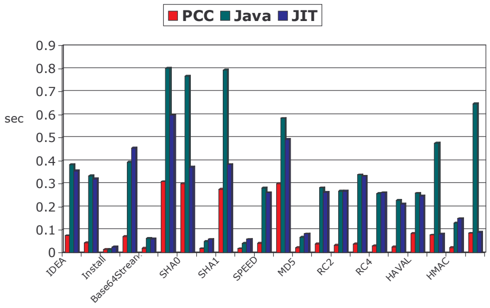


Reasonable in size (0-10%).

Simple, small (<52KB), and fast.

No longer need to trust this component.

Crypto test suite results



Question 3

Aren't the properties we're trying to prove undecideable?

How on earth can we hope to generate the proofs?

How to generate the proofs?

Proving theorems about real programs is indeed hard

- Most useful safety properties of low-level programs are undecidable
- Theorem-proving systems are unfamiliar to programmers and hard to use even for experts

The role of programming languages

Civilized programming languages can provide “safety for free”

- Well-formed/well-typed ⇒ safe

Idea: Arrange for the compiler to “explain” why the target code it generates preserves the safety properties of the source program

Certifying Compilers

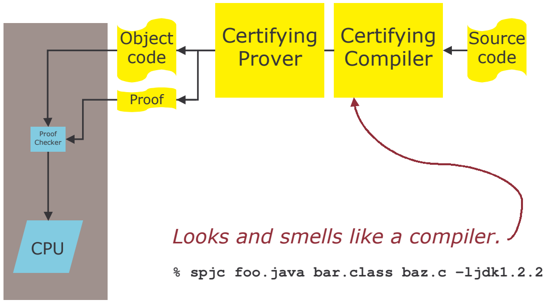
[Necula & Lee, PLDI’98]

Intuition:

- Compiler “knows” why each translation step is semantics-preserving
- So, have it generate a proof that safety is preserved

This is the planned topic for tomorrow’s lecture

Certifying compilation



Looks and smells like a compiler.

```
% spjc foo.java bar.class baz.c -ljdk1.2.2
```

Java

Java is a worthwhile subject of research.

However, it contains many *outrageous* and mostly *inexcusable* design errors.

As researchers, we should not forget that we have already done much better, and must continue to do better in the future.

Question 4

Just what, exactly, are we proving?

What are the limits?

And isn’t static checking inherently less powerful than dynamic checking?

Semantics

Define the states of the target machine

- $S = (\underset{\text{program}}{\Pi}, \underset{\text{register state}}{\rho}, \underset{\text{program counter}}{pc})$

and a transition function $\text{Step}(S)$.

Define also the safe machine states via the *safety policy* $\text{SP}(S)$.

Semantics, cont’d

Then we have the following predicate for safe execution:

$$\text{Safe}(S) = \Pi n{:}\text{Nat}.\ \text{SP}(\text{Step}^{n}(S))$$

and proof-carrying code:

$$\text{PCC} = (S_0{:}\text{State},\ \text{P}{:}\text{Safe}(S_0))$$

Reference Interpreters

A *reference interpreter (RI)* is a standard interpreter extended with instrumentation to check the safety of each instruction before it is executed, and abort execution if anything unsafe is about to happen.

In other words, an RI is capable *only* of safe execution.

Reference Interpreters
cont’d

The reference interpreter is never actually implemented.

The point will be *to prove that execution of the code on the RI never aborts, and thus execution on the real hardware will be identical to execution on the RI*.

Question for you

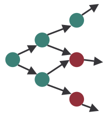
Suppose that we require the code to execute no more than *N* instructions.

Is such a safety property enforceable by an RI?

Question for you

Suppose we require the code to terminate eventually. Is such a safety property enforceable by an RI?

What can’t be enforced?



Informally:

Safety properties ⇒ *Yes*

- *“No bad thing will happen”*

Liveness properties ⇒ *Not yet*

- *“A good thing will eventually happen”*

Static vs dynamic checking

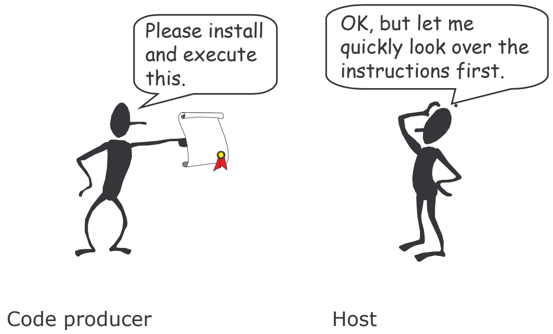
PCC provides a basis for static enforcement of safety conditions

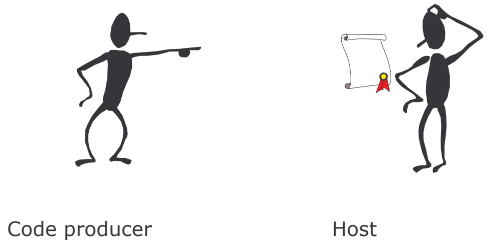
However, PCC is not just for static checking

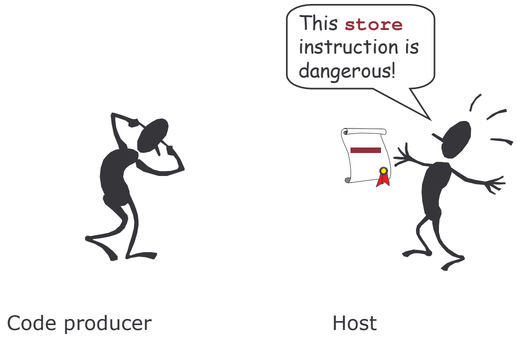
PCC can be used, for example, to verify that necessary dynamic checks are carried out properly

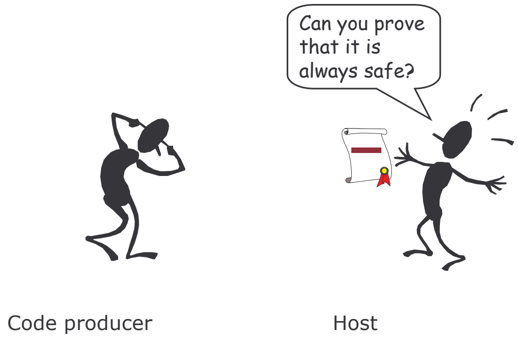
Question 5

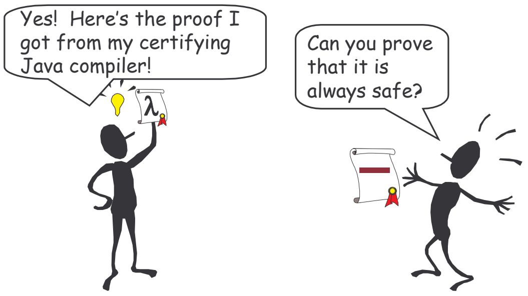
Even if the proof is valid, how do we know that it is a safety proof of the given program?



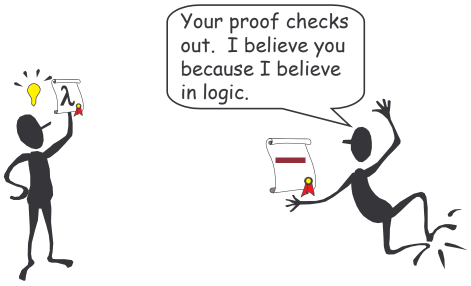








Code producer Host



Code producer Host

The safety policy

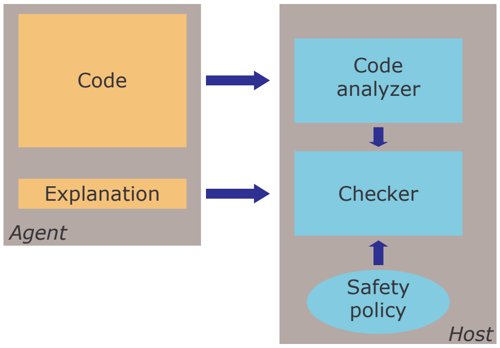
We need a method for

- identifying the dangerous instructions, and
- generating logical predicates whose validity implies that the instruction is safe to execute

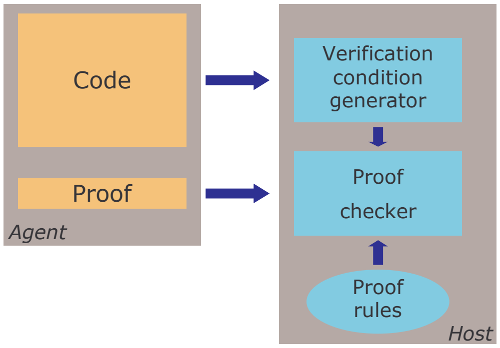
In practice, we will also need

- specifications (pre/post-conditions) for each required entry point in the code, as well as the trusted API.

High-level architecture



High-level architecture



VCgen

The job of identifying dangerous instructions and generating predicates for them is performed via an old method:

- *verification-condition generation*

A Case Study

A case study

As a case study, let us consider the problem of verifying that programs do not use more than a specified amount of some resource.

```
s ::= skip
    | i := e
    | if e then s else s
    | while e do s
    | s ; s
    | use e

e ::= n
    | i | read()
    | e + e | e − e | …
```

Denotes the use of n pieces of the resource, where e evaluates to n

Case study, cont'd

Under normal circumstances, one would implement the statement:

- use e;

in such a way that every time it is executed, a run-time check is performed in order to determine whether *n* pieces of the resource are available (assuming e evaluates to *n*).

Case study, cont'd

However, this stinks because many times we should be able to infer that there are definitely available resources.

If somehow we know that there are ≥9 available here…

```
…
if …
  then use 4;
  else use 5;
use 4;
…
```

…then certainly there is no need to check any of these uses!

An easy (well, probably) case

```
Program Static
  i := 0
  while i < 10000
    use 1
    i := i + 1
```

We ought to be able to prove statically whether the uses are safe

A hopeless case

```
Program Dynamic
  while read() != 0
    use 1
```

An interesting case

```
Program Interesting
  N := read()
  i := 0
  while i < N
    use 1
    i := i + 1
```

In principle, with just a single dynamic check, static proof ought to be possible

Also interesting

```
Program AlsoInteresting
  while read() != 0
    i := 0
    while i < 100
      use 1
      i := i + 1
```

A core principle of PCC

In the code,

- the implementation of a safety-critical operation

should be *separated* from

- the implementation of its safety checks

Separating use from check

So, what we would like to do is to separate the safety check from the use.

We do this by introducing a new construct, **`acquire`**

`acquire` requests n amount of resource; **`use`** no longer does any checking

```
s ::= skip
    | i := e
    | if e then s else s
    | while e do s
    | s ; s
    | use e
    | acquire e
```

Separation permits optimization

The point of **`acquire`** is to allow the programmer (or compiler) to hoist and coalesce the checks

```
…
acquire 9;
if …
  then use 4;
  else use 5;
use 4;
…
```

```
…
acquire n;
i := 0;
while (i++ < n) do {
  …
  use 1;
  …
}
```

It will be up to PCC to verify that each use is definitely safe to execute

High-level architecture