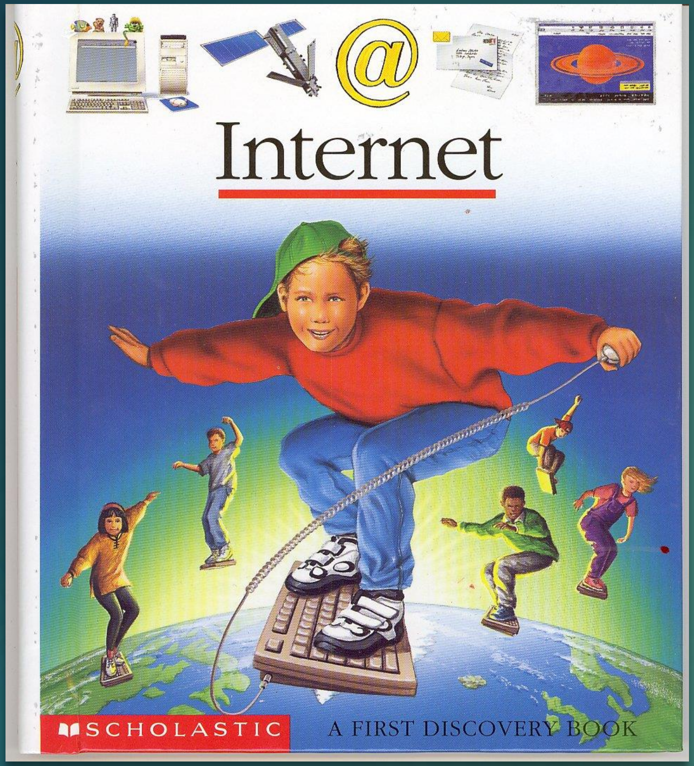
Proof-Carrying Code

GEORGE C. NECULA, POPL '97

PRESENTED BY TOM MAGRINO AND MENTORED BY ETHAN CECCHETTI IN GREAT WORKS IN PL, APRIL 16TH, 2019



"On the Internet, nobody knows you're a dog."

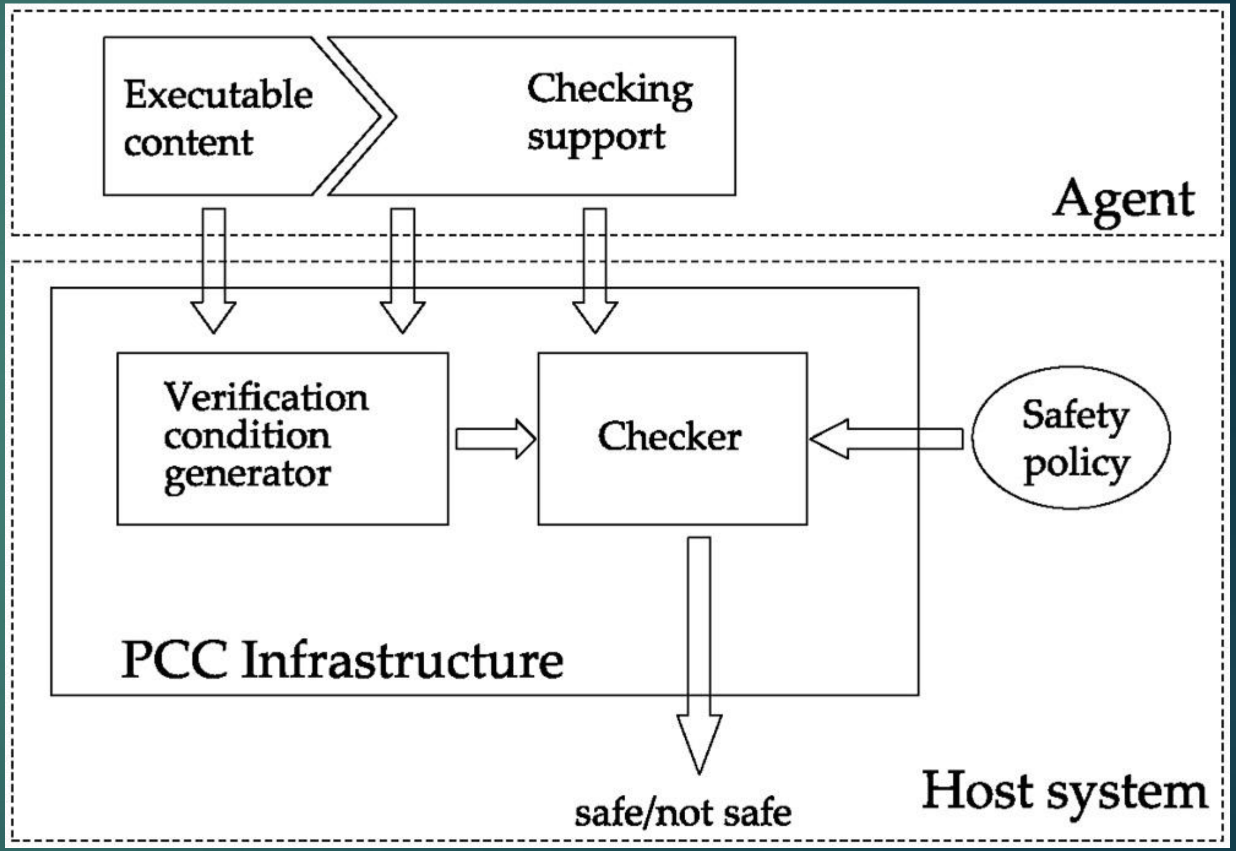
How can you trust that code you downloaded?

Context

- Similar motivation to TAL: Want user-supplied code that can run in sensitive contexts (e.g. in the kernel, in a host process, etc.) with assurance that some properties hold.
  - Packet filtering (Necula & Lee OSDI '96)
  - Libraries implemented in another language
  - Mobile code (e.g., JavaScript)
- Techniques prior:
  - Specialized DSLs
    - Limited expressions and yet-another-language to learn
  - Runtime monitors
    - Runtime overhead
  - Compile on demand
    - Compile time overhead

Core Idea

- Ship machine code with a simple, verifiable proof of desired properties.
- Programmer or compiler creates proof, which is attached to the binary.
- Host validates the proof before running it the first time.
  - When sent already validated code, just verify it's the same proof.



Safety Policies

- Safety Policy:
  - Language of symbolic expressions and formulas for verification conditions.
  - Set of pre- and postconditions for all interface functions between host and agent.
  - Set of proof rules for verification conditions.

Case Study: Safe Extension to ML
"Safe Sum"

```
datatype T = Int of int | Pair of int * int

fun sum (l : T list) =
 let
  fun foldr f nil a = a
    | foldr f (h::t) a = foldr f t (f(a, h))
 in
  foldr (fn (acc, Int i) => acc + i
            | (acc, Pair (i, j)) => acc + i + j)
         l 0
 end
```

```
                                          %r0 is l
   sum : INV  rm ⊢ r0 : T list
                                          %r1 is acc
         MOV  r1, 0                       %Initialize acc
   L2    INV  rm ⊢ r0 : T list ∧ rm ⊢ r1 : int
                                          %Loop invariant
         BEQ  r0, L14                     %Is list empty?
         LD   r2, 0(r0)                   %Load head
         LD   r0, 4(r0)                   %Load tail
         LD   r3, 0(r2)                   %Load constructor
         LD   r2, 4(r2)                   %Load data
         BEQ  r3, L12                     %Is an integer?
         LD   r3, 0(r2)                   %Load i
         LD   r2, 4(r2)                   %Load j
         ADD  r2, r3, r2                  %Add i and j
   L12   ADD  r1, r2, r1                  %Do the addition
         BR   L2                          %Loop
   L14   MOV  r0, r1                      %Copy result in r0
         RET                              %Result is in r0
```

- Policy: program respects type-safety and calling conventions.
  - References are only to valid memory locations
  - Postcondition is satisfied (result is left in the appropriate register with correct type).

$$Pre \equiv \mathbf{r_m} \vdash \mathbf{r_0} : \mathrm{T\ list}$$
$$Post \equiv \mathbf{r_m} \vdash \mathbf{r_0} : \mathrm{int}$$

Proving Correctness: Type Rules

- Typing Rules: m ⊦ e : τ
  - m – memory State (types for a subset of addresses)
  - e – expression in assembly
  - τ – type of expression
- e ::= n | r_i | sel(m, e) | e_1 + e_2
- m ::= r_m | upd(m, e_1, e_2)

Pair
$$\frac{m \vdash e : \tau_1 * \tau_2}{m \vdash e : \mathbf{addr} \;\wedge\; m \vdash e+4 : \mathbf{addr} \;\wedge\; m \vdash \mathbf{sel}(m,e) : \tau_1 \;\wedge\; m \vdash \mathbf{sel}(m,e+4) : \tau_2}$$

Sum
$$\frac{m \vdash e : \tau_1 + \tau_2}{m \vdash e : \mathbf{addr} \;\wedge\; m \vdash e+4 : \mathbf{addr} \;\wedge\; \mathbf{sel}(m,e) = 0 \supset m \vdash \mathbf{sel}(m,e+4) : \tau_1 \;\wedge\; \mathbf{sel}(m,e) \neq 0 \supset m \vdash \mathbf{sel}(m,e+4) : \tau_2}$$

List
$$\frac{m \vdash e : \tau\ \mathbf{list} \qquad e \neq 0}{m \vdash e : \mathbf{addr} \;\wedge\; m \vdash e+4 : \mathbf{addr} \;\wedge\; m \vdash \mathbf{sel}(m,e) : \tau \;\wedge\; m \vdash \mathbf{sel}(m,e+4) : \tau\ \mathbf{list}}$$

Int
$$\frac{m \vdash e_1 : \mathbf{int} \qquad m \vdash e_2 : \mathbf{int}}{m \vdash e_1 + e_2 : \mathbf{int}} \qquad \frac{}{m \vdash 0 : \mathbf{int}}$$

Verification Conditions

- Approach: create conditions for each instruction.
- Top-level: "For all register values, every invariant implies the condition of the next instruction."

$$VC_i = \begin{cases} [\mathbf{r}_s + op/\mathbf{r}_d]\, VC_{i+1}, & \textit{if } \Pi_i = \text{ADD } \mathbf{r}_s, op, \mathbf{r}_d \\ \mathbf{r_m} \vdash \mathbf{r}_s + n : \mathbf{addr} \wedge [\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s + n)/\mathbf{r}_d]\, VC_{i+1}, & \textit{if } \Pi_i = \text{LD } \mathbf{r}_d, n(\mathbf{r}_s) \\ (\mathbf{r}_s = 0 \supset VC_{i+n+1}) \wedge (\mathbf{r}_s \neq 0 \supset VC_{i+1}), & \textit{if } \Pi_i = \text{BEQ } \mathbf{r}_s, n \\ Post, & \textit{if } \Pi_i = \text{RET} \\ \mathcal{I}, & \textit{if } \Pi_i = \text{INV } \mathcal{I} \end{cases}$$

$$VC(\Pi, Inv, Post) = \forall \mathbf{r}_i. \bigwedge_{i \in Inv} Inv_i \supset VC_{i+1}$$

For Example: $\mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{Foo list} \supset (\mathbf{r_m} \vdash \mathbf{r}_0 : \texttt{Foo list} \wedge \mathbf{r_m} \vdash 0 : \texttt{int})$

Constructing a Safety Proof

- Use a logic framework (LF) to encode the proof of the desired property.
  - Meta-language for specifications of logics
- Proof becomes a program in LF and validation is type-checking the proof has type pf Post.

$$\texttt{and\_i} \;:\; \Pi p{:}\texttt{pred}.\Pi r{:}\texttt{pred}.\; \texttt{pf}\; p \to \texttt{pf}\; r \to \texttt{pf}\; (\texttt{and}\; p\; r)$$

$$\ulcorner \dfrac{\begin{matrix} \mathcal{D}_1 & & \mathcal{D}_2 \\ \triangleright P_1 & & \triangleright P_2 \end{matrix}}{\triangleright P_1 \wedge P_2} \urcorner = \texttt{and\_i}\; \ulcorner P_1 \urcorner\; \ulcorner P_2 \urcorner\; \ulcorner \mathcal{D}_1 \urcorner\; \ulcorner \mathcal{D}_2 \urcorner$$

Constructing a Safety Proof

- Use a logic framework (LF) to encode the proof of the desired property.
  - Meta-language for specifications of logics
- Proof becomes a program in LF and validation is type-checking the proof has type pf Post.

$$\frac{m \vdash e : \tau\ \mathtt{list} \qquad e \neq 0}{m \vdash e : \mathtt{addr}\ \wedge\ m \vdash e+4 : \mathtt{addr}\ \wedge\ m \vdash \mathtt{sel}(m,e) : \tau\ \wedge\ m \vdash \mathtt{sel}(m,e+4) : \tau\ \mathtt{list}}$$

```
tp_list : Πm:exp.Πe:exp.Πt:tp.
  pf (hastype m e (list t)) → pf (neq e 0) →
  pf (and (and (hastype m e addr)
               (hastype m (sel m e) t))
          (and (hastype m (+ e 4) addr)
               (hastype m (sel m (+ e 4)) (list t))))
```

Quick Aside: Encoding Proofs

- Implicit LF: Avoid redundant terms in encoded proof.
  - Extends LF with placeholders for redundant proof terms.
  - Reused proofs don't require redundant checks!
  - Custom algorithm for reconstructing the terms for placeholders during type-checking.
    - Requires adding rules not directly useful for type checking or type inference.
- See Ch. 5 of Advanced Topics in TaPL for more!

PCC in Practice

- Proof ships with the program, gets verified by the host, and we're ready to go.
- Sum example code: 730 bytes
  - Proof: 420 bytes
  - Code: 60 bytes
  - "Fixed-sized Overhead": 250 bytes
- Validation (on 175 MHz machine) was 1.9ms
  - On a modern processor this translates to microseconds.
- Packet Filters
  - Showed 10x improvement over runtime checking.
  - Allowed user defined code in the kernel with safety guarantees.

Takeaways of PCC

- PL technique to solve important engineering problem!
  - Maybe obvious to us, was a big deal for systems and security.
- Generalizes beyond traditional types:
  - Security policies.
  - Concurrency rules.
  - Domain-specific safety rules.
- Small trusted computing base (TCB) for important class of security problems.
  - TCB = checker + any tools that generate the proofs (for honest users).
- Kicked off a huge line of work!

Discussion

- Where do we see this in today's systems?
- How does this compare/contrast with TAL?
- Do modern techniques make annotations and proofs easier to produce?
- Potential new application domains?