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
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
```

- 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  ≡ r_m ⊢ r_0 : T list
Post ≡ r_m ⊢ r_0 : int
```
Proving Correctness: Type Rules

- Typing Rules: $m \vdash e : \tau$
  - $m$ – memory State (types for a subset of addresses)
  - $e$ – expression in assembly
  - $\tau$ – type of expression
- $e ::= n \mid r \mid \text{sel}(m, e) \mid e_1 + e_2$
- $m ::= r \mid \text{upd}(m, e_1, e_2)$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair</td>
<td>$m \vdash e : \tau_1 \cdot \tau_2 \quad m \vdash e : \text{addr} \wedge m \vdash e + 4 : \text{addr} \wedge m \vdash \text{sel}(m, e) : \tau_1 \wedge m \vdash \text{sel}(m, e + 4) : \tau_2$</td>
<td>$m \vdash e : \text{addr}$</td>
</tr>
<tr>
<td>Sum</td>
<td>$m \vdash e : \text{addr} \wedge m \vdash e + 4 : \text{addr} \wedge \text{sel}(m, e) = 0 \supset m \vdash \text{sel}(m, e + 4) : \tau_1 \wedge \text{sel}(m, e) \neq 0 \supset m \vdash \text{sel}(m, e + 4) : \tau_2$</td>
<td>$m \vdash e : \tau_1 + \tau_2$</td>
</tr>
<tr>
<td>List</td>
<td>$m \vdash e : \tau \text{ list}$</td>
<td>$m \vdash e : \tau$</td>
</tr>
<tr>
<td>Int</td>
<td>$m \vdash e_1 : \text{int} \quad m \vdash e_2 : \text{int}$</td>
<td>$m \vdash e_1 + e_2 : \text{int}$</td>
</tr>
<tr>
<td></td>
<td>$m \vdash 0 : \text{int}$</td>
<td></td>
</tr>
</tbody>
</table>
Verification Conditions

- Approach: create conditions for each instruction.

- Top-level: “For all register values, every invariant implies the condition of the next instruction.”

\[
VC_i = \begin{cases} 
[r_s + op/r_d] VC_{i+1}, \\
\mathit{r}_m \vdash r_s + n : \mathit{addr} \land [\mathit{sel}(\mathit{r}_m, r_s + n)/r_d] VC_{i+1}, \\
(r_s = 0 \supset VC_{i+n+1}) \land (r_s \neq 0 \supset VC_{i+1}), \\
Post, \\
I, 
\end{cases}
\]

For Example:

\[
VC(\Pi, \mathit{Inv}, \mathit{Post}) = \forall r_i. \bigwedge_{i \in \mathit{Inv}} \mathit{Inv}_i \supset VC_{i+1}
\]
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.

\[
\begin{align*}
\text{and}_i & : \Pi p: \text{pred.} \Pi r: \text{pred.} \\
& \quad \text{pf } p \rightarrow \text{pf } r \rightarrow \text{pf } (\text{and } p \ r)
\end{align*}
\]
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.

\[
\begin{align*}
 & m \vdash e : \tau \text{ list} \\
 & m \vdash e : \text{addr} \land m \vdash e + 4 : \text{addr} \land m \vdash \text{sel}(m, e) : \tau \land m \vdash \text{sel}(m, e + 4) : \tau \text{ list}
\end{align*}
\]

\[
\begin{align*}
 & \text{tp_list} : \Pi m: \text{exp.} \Pi e: \text{exp.} \Pi t: \text{tp.} \\
 & \quad \begin{aligned}
 & \text{pf (hastype } m \ e \ (\text{list } t)) \rightarrow \text{pf (neq } e \ 0) \rightarrow \\
 & \quad \text{pf (and (and (hastype } m \ e \ \text{addr})
 & \qquad (\text{hastype } m \ (\text{sel } m \ e) \ t))
 & \qquad (\text{hastype } m \ (+ \ e \ 4) \ \text{addr})
 & \qquad (\text{hastype } m \ (\text{sel } m \ (+ \ e \ 4)) \ (\text{list } t)))
\end{aligned}
\end{align*}
\]
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?