Proof-Carrying Code

Chapter by George Necula
Presented by Brian Corcoran
Overview

• General framework that allows the host to quickly and easily check that an agent has certain safety properties

• Agent producer cooperates with host by attaching a proof of why the code complies with the safety property to the agent code

• George Necula’s dissertation (1998)
Implementations of PCC Concept

- JVML and CLI verifiers
- Typed assembly language (TAL)
- Foundational PCC
- **Touchstone PCC**
Touchstone PCC Architecture
Verification-Condition Generator (VCGen)

- Directly checks simple syntactic conditions
- Possibly unsafe instructions are passed to the Checker module
- “Compiles” programs to logical formulas
VCGen

- In some cases, needs to understand complex invariants in agent code
- Control flow & data flow
- Relies on code annotations from agent
- Moves burden to agent producer
Checker

- Verifies that all dangerous instructions are used in a safe context
- Uses verification conditions generated by VCGen and proof from agent to check that safety preconditions are met
Safety Policy

- Customizable collection of configuration data that specifies logic for VCGen and trusted proof rules that can be used in safety proofs for an agent producer

- Example: untrusted code must interact correctly with runtime of JVM
Example Agent

• Check a simple type-safety policy for an agent written in a generic assembly language

• Adds all the elements of a list containing either integers or pairs of integers
type maybepair = Int of int | Pair of int * int
let rec sum(acc : int, x : maybepair list) =
  match x with
  | nil → acc
  | (Int i) :: tail → sum(acc + i, tail)
  | (Pair (l, r)) :: tail → sum(acc + l + r, tail)
Representation Strategy

• List:
  • value 0, or pointer to two-word memory

• maybepair:
  • Pair(x,y) as even-valued pointer to two-word memory
  • Int(x) as integer 2x + 1

• Example: [Int 2; Pair (3, 4)]
\[ r_x := e \quad \text{assign the result of } e \text{ to register } r_x \]
\[ r_x := \text{Mem}[e] \quad \text{load } r_x \text{ from address } e \]
\[ \text{Mem}[e'] := e \quad \text{store the result of } e \text{ to address } e' \]
\[ \text{jump } L \quad \text{jump to a label } L \]
\[ \text{if } e \text{ jump } L \quad \text{branch to label } L \text{ if } e \text{ is true} \]
\[ \text{return } \quad \text{return from the current function} \]
sum:

Loop:

3 if \( r_x \neq 0 \) jump LCons

4 \( r_R := r_{acc} \)

5 return

6 LCons: \( r_t := \text{Mem}[r_x] \)

7 if even(\( r_t \)) jump LPair

8 \( r_t := r_t \div 2 \)

9 \( r_{acc} := r_{acc} + r_t \)

10 jump LTail

11 LPair: \( r_s := \text{Mem}[r_t] \)

12 \( r_{acc} := \text{Mem}[r_{acc} + r_s] \)

13 \( r_t := \text{Mem}[r_t + 4] \)

14 \( r_{acc} := r_{acc} + r_t \)

15 LTail: \( r_x := \text{Mem}[r_x + 4] \)

16 jump Loop

; \( r_x \) : maybepair list

; Is \( r_x \) empty?

; Load the first data

; Get the first pair element

; and the second element

Assembly code for example function
Safety Policies

• Memory reading & writing
• Agent assumptions
Memory Read Policy

- From pointers that are either non-null lists
- Read either the first or second field of a list
- From pointers to elements of the Pair kind
Memory Write Policy

- First word of a list cell can be either an odd value, or even value that is a pointer to an element of Pair kind
- Second word can be either zero or pointer to a list cell
- Ensures that content of accessible memory locations is consistent with type
Policy Assumptions

• Safety policy also specifies assumptions the agent can make about execution context

• On entry, contents of register $r_x$ is either zero or a pointer to a list cell

• Value read from list cell is either odd or a pointer to a Pair cell

• Value read from second word of list cell is either 0 or a pointer to a list cell
Formalizing Safety Policy

• List of instructions whose execution may violate safety

• For each, specifies the verification condition that guarantees safe execution

• In this example:
  • memory operations
  • function calls and returns
Safety Policy

• Choice of instructions is hard-coded
• Sufficient to encode a large class of safety properties
• Later, show safety policy not covered
  • Requires change to VCGen
Logic Syntax

Formulas \( F ::= \text{true} | F_1 \land F_2 | F_1 \lor F_2 | F_1 \Rightarrow F_2 | \forall x.F | \exists x.F \\
| \text{addr } E_a | E_1 = E_2 | E_1 \neq E_2 | f E_1 \ldots E_n \)

Expressions \( E ::= x | \text{sel } E_m E_a | \text{upd } E_m E_a E_v | f E_1 \ldots E_n \)

- First-order language of symbolic expressions and formulas
- Expressions: variables and constructors (integers, arithmetic operators)
Customizable Elements

- Language of symbolic expressions and formulas to express verification conditions
- Function pre- and postconditions
- Forms interface between host and agent
- Proof rules for verification conditions
Syntax Extensions

• Safety policy extends syntax by defining new expression and formula constructors

• Add constructors for types and predicate constructor for encoding typing judgment

Word types \( W ::= \text{int} \mid \text{ptr}\{S\} \mid \text{list}\ W \mid \{x \mid F(x)\} \)

Structure types \( S ::= W \mid W;S \)

Formulas \( F ::= \ldots \mid E:W \mid \text{listinv } E_m \)
listinv

formula constructor

• States that the contents of memory satisfies invariants for list pairs

• “listinv \( M \) holds when each memory address is assigned a pointer type that contains values that are assigned appropriate types”
Pre- and Postconditions

- Forms interface between agent and host
- Functions defined by agent and invoked by host
- Library functions exported by host
- Expressed in terms of registers
- Model memory as pseudo-register $r_M$
Pre- and Postconditions

• For our example:

\[
\begin{align*}
\text{Pre}_{\text{sum}} &= r_x : \text{mp\_list} \land \text{listinv } r_M \\
\text{Post}_{\text{sum}} &= \text{listinv } r_M
\end{align*}
\]

• Memory state must be well-typed after the agent returns

• Agent can assume that memory and \( r_x \) are well-typed when host invokes it
Proof Rules

• Set of proof rules to reason about verification conditions

• (Select) Built-in proof rules:

\[
\begin{array}{c|c}
F_1 & F_2 \\
\hline
F_1 \land F_2 & (\text{ANDI}) \\
F_1 \land F_2 & (\text{ANDEL}) \\
F_2 & (\text{ANDER}) \\
\hline
F_1 & \\
\vdots & \\
F_1 \Rightarrow F_2 & (\text{IMPI}) \\
F_1 \Rightarrow F_2 & F_1 \\
F_2 & (\text{IMPE}) \\
A = A' & (\text{MEM0}) \\
\text{sel} (\text{upd } M \ A \ V) \ A' = V \\
A \neq A' & \\
\text{sel} (\text{upd } M \ A \ V) \ A' = \text{sel } M \ A' & (\text{MEM1}) \\
\end{array}
\]
Extending Proof Rules

- Safety policy extends proof rules:

\[
\begin{align*}
0 : \text{list } W \\
E : \text{list } W & \quad E \neq 0 \\
\overline{E : \text{ptr } \{W;\text{list } W\}} & \quad \text{(NIL)} \\
E : \{y \mid F(y)\} & \quad F(E) \\
\overline{E : \text{ptr } \{W;S\}} & \quad \text{(CONS)} \\
E : \text{ptr } \{W\} & \quad \text{(SET)} \\
\end{align*}
\]

\[
\begin{align*}
E : \text{ptr } \{W;S\} & \quad E + 4 : \text{ptr } \{S\} & \quad \text{(NEXT)} \\
A : \text{ptr } \{W\} & \quad \text{listinv } M & \quad \text{(SEL)} \\
\overline{(\text{sel } M A) : W} & \quad \text{(TP)} \\
\text{listinv } M & \quad A : \text{ptr } \{W\} & \quad V : W & \quad \text{(UPD)} \\
\overline{\text{listinv } (\text{upd } M A V)} & \quad \text{(PTRADDR)} \\
A : \text{ptr } \{W\} & \quad \text{addr } A & \quad \text{(PTRADDR)}
\end{align*}
\]

- \text{PTRADDR} : relates new constructors with built-in \text{addr} memory safety constructor
Verification-Condition Generation

- How do we enforce safety policy?
- Analogous to high-level type systems
  - Have type system and rules; need type checker
- VCGen scans code, decides what rules to apply, and what needs to be checked
Hard to Apply Proof Rules

- Hard to use pattern matching
- Sensitive to code generation and optimization
- High-level operations split into several small operations
- Algorithm must be flow and path sensitive
- Variables do not have single type throughout scope
Approaches : JVML

• Bundle complex operations in high-level constructs

• Ensures outcome of conditionals does not effect type checking

• Simplifies type-checking problem

• Agent producer cannot do much (compile-time) optimization

• Puts burden on code receiver for compiling and optimization
Symbolic Execution

- Originally used in program verification
- Can verify full correctness, not just well-typedness
Symbolic Execution

- Consider example without conditional
- Reuses registers with different types
- Splits high-level operations into multiple low-level instructions
- More important to remember effect of instructions vs. checking immediately
- Postpone checking as much as possible
Symbolic Execution

- If we allow arbitrary complex operands:
  - $r_t := \text{Mem}[r_x + 4]$
- Now use pattern matching to choose rule!
Limitations of SE

• In some cases, symbolic evaluator should not directly follow control flow:
  • Loops
  • Functions

• In these cases, we need assistance from code producer, in form of code annotations
Program Annotations

- Makes symbolic execution possible in finite time without conservative approximations
- At least one invariant for each cycle in control-flow graph
- Example: Loop: \( \text{INV} = r_x : \text{mp_list} \land \text{listinv r}_M \)
1 sum: \( \text{INV } r_x : \text{mp_list} \land \text{listinv } r_M \)
2 Loop: \( \text{INV } r_x : \text{mp_list} \land \text{listinv } r_M \)
3 \( \text{if } r_x \neq 0 \text{ jump LCons} \) ; list is empty
4 \( r_R := r_{acc} \)
5 return
6 LCons: \( r_t := \text{Mem}[r_x] \) ; Load the first data
7 \( \text{if } \text{even}(r_t) \text{ jump LPair} \)
8 \( r_t := r_t \div 2 \)
9 \( r_{acc} := r_{acc} + r_t \)
10 jump LTail
11 LPair: \( r_s := \text{Mem}[r_t] \) ; Get the first pair element
12 \( r_{acc} := \text{Mem}[r_{acc} + r_s] \)
13 \( r_t := \text{Mem}[r_t + 4] \) ; and the second element
14 \( r_{acc} := r_{acc} + r_t \)
15 LTail: \( r_x := \text{Mem}[r_x + 4] \)
16 jump Loop

Assembly code for example function
Creating Annotations

• Where do annotations come from?
  • Inserted by hand
  • Produced automatically by a certifying compiler
  • Invariant annotations cannot be trusted!
Verification-Condition Generator

- Assume we have one function with pre- and postcondition specified by safety policy
- Use symbolic evaluation function SE
- Takes values $i, \sigma$
- Produces all verification conditions from the given PC until the next return instruction or invariant
\[
SE(i, \sigma) = \begin{cases} 
SE(i + 1, \sigma[r \leftarrow \sigma e]) & \text{if } \Pi_i = r := e \\
(\sigma e) \Rightarrow SE(L, \sigma) \land \\
(\text{not} (\sigma e)) \Rightarrow SE(i + 1, \sigma) & \text{if } \Pi_i = \text{if } e \text{ jump } L \\
\text{addr} (\sigma a) \land \\
SE(i + 1, \sigma[r \leftarrow (\sigma (\text{sel } r_M a))]) & \text{if } \Pi_i = r := \text{Mem}[a] \\
\text{addr} (\sigma a) \land \\
SE(i + 1, \sigma[r_M \leftarrow (\sigma (\text{upd } r_M a e))]) & \text{if } \Pi_i = \text{Mem}[a] := e \\
\sigma \text{ Post} & \text{if } \Pi_i = \text{return} \\
\sigma I & \text{if } \Pi_i = \text{INV } I \\
\end{cases}
\]

Definition of SE Function
1: Generate fresh values $r_M = m_0$, $r_R = r_0$, $r_X = x_0$, $r_{acc} = acc_0$,
   $r_t = t_0$ and $r_s = s_0$

1: Assume Invariant
   \[
   x_0: \text{mp_list} \\
   \text{listinv } m_0
   \]

2: Invariant
   \[
   x_0: \text{mp_list} \\
   \text{listinv } m_0
   \]

2: Generate fresh values $r_M = m_1$, $r_R = r_1$, $r_X = x_1$, $r_{acc} = acc_1$,
   $r_t = t_1$ and $r_s = s_1$

2: Assume Invariant
   \[
   x_1: \text{mp_list} \\
   \text{listinv } m_1
   \]

3: Branch 3 taken
   \[
   x_1 \neq 0
   \]

6: Check load
   \[
   \text{addr } x_1
   \]

7: Branch 7 taken
   \[
   \text{even } \langle \text{sel } m_1 \times x_1 \rangle
   \]

11: Check load
    \[
    \text{addr } \langle \text{sel } m_1 \times x_1 \rangle
    \]

13: Check load
    \[
    \text{addr } \langle \text{sel } m_1 \times x_1 \rangle + 4
    \]

15: Check load
    \[
    \text{addr } (x_1 + 4)
    \]

16: Goto Loop
2: Invariant
    \[
    \langle \text{sel } m_1 (x_1 + 4) \rangle: \text{mp_list} \\
    \text{listinv } m_1
    \]

7: Branch 7 not taken
   \[
   \text{odd } \langle \text{sel } m_1 \times x_1 \rangle
   \]

10: Goto LTail
15: Check load
    \[
    \text{addr } (x_1 + 4)
    \]

16: Goto Loop
2: Invariant
    \[
    \langle \text{sel } m_1 (x_1 + 4) \rangle: \text{mp_list} \\
    \text{listinv } m_1
    \]

3: Branch 3 not taken
   \[
   x_1 = 0
   \]

5: Return
   \[
   \text{listinv } m_1
   \]
Proof of a verification condition
Soundness Proof

- “If the global verification condition for a program is provable using the proof rules given by the safety policy, then the program is guaranteed to execute without violating memory safety.”
• Formally defines small-step operation semantics of assembly language

• Splits proof into safety policy rules, VCGen

• Proves soundness of VCGen

• Rules out possibility that execution gets stuck from executing an instruction at an invalid program counter or tries to dereference an invalid address
\[\models_{\mathcal{M}} F_1 \land F_2 \quad \text{iff} \quad \models_{\mathcal{M}} F_1 \text{ and } \models_{\mathcal{M}} F_2\]
\[\models_{\mathcal{M}} F_1 \Rightarrow F_2 \quad \text{iff} \quad \text{whenever } \models_{\mathcal{M}} F_1 \text{ then } \models_{\mathcal{M}} F_2\]
\[\models_{\mathcal{M}} \forall x.F(x) \quad \text{iff} \quad \forall e \in \mathbb{Z}. \models_{\mathcal{M}} F(e)\]

\[
\models_{\mathcal{M}} a: \text{int} \quad \text{iff} \quad a \in \mathbb{Z} \\
\models_{\mathcal{M}} a: \text{list } W \quad \text{iff} \quad a = 0 \lor (\mathcal{M}(a) = W \land \mathcal{M}(a + 4) = \text{list } W) \\
\models_{\mathcal{M}} a: \text{ptr } \{S\} \quad \text{iff} \quad \forall i.0 \leq i < |S| \Rightarrow \mathcal{M}(a + 4 \times i) = S_i \\
\models_{\mathcal{M}} a: \{y | F(y)\} \quad \text{iff} \quad \models_{\mathcal{M}} F(a) \\
\models_{\mathcal{M}} \text{listinv } m \quad \text{iff} \quad \forall a \in \text{Dom}(\mathcal{M}).a \in \text{Dom}(m) \text{ and } \models_{\mathcal{M}} m a : \mathcal{M}(a) \\
\models_{\mathcal{M}} \text{addr } a \quad \text{iff} \quad a \in \text{Dom}(\mathcal{M})
\]

Soundness of the safety property
Abstract machine for soundness proof

(i, \rho) \rightsquigarrow \begin{cases} 
(i + 1, \rho[r_d \leftarrow \rho e]), & \text{if } \Pi_i = \text{set } r_d \text{ to } e \\
(i + 1, \rho[r_d \leftarrow \rho (sel \ r_M e)]), & \text{if } \Pi_i = \text{load } r_d \text{ from } e \\
& \quad \text{and } \rho e \in Addr \\
(i + 1, \rho[r_M \leftarrow \rho (upd \ r_M e_2 e_1)]), & \text{if } \Pi_i = \text{write } e_1 \text{ to } e_2 \\
& \quad \text{and } \rho e_2 \in Addr \\
(L, \rho), & \text{if } \Pi_i = \text{if } e \text{ goto } L \\
& \quad \text{and } \rho e \\
(i + 1, \rho), & \text{if } \Pi_i = \text{if } e \text{ goto } L \\
& \quad \text{and } \rho (\text{not } e) \\
(i + 1, \rho), & \text{if } \Pi_i = \text{INV } I
\end{cases}
Representation and Checking of Proofs

- Soundness theorem requires agent meet safety policy
- Verify rules by witnessing a derivation using sound system of proof rules
- Attach derivation to untrusted code so Checker can find and check it
Framework Requirements

- Want framework of encoding proofs of logical formulas that is:
  - General (not specific to particular logic)
  - Relatively compact
  - Easy to check
Edinburgh Logical Framework (LF)

• Simple variant of the \( \lambda \)-calculus

• If predicate is an LF type, then any LF expression of that type is a proof of that predicate

• Proof of \( F \Rightarrow (F \wedge F) \):
  \[
  M = \text{impi } 'F' (\text{and } 'F' 'F')
  \]
  \[
  (\lambda x:pf 'F'.\text{andi } 'F' 'F' x x)
  \]

• Allows implementation that is simple, easy to trust, and allows logic to be extended
Problems with LF

- Typical LF representation is large due to significant redundancy
- Redundancy increases non-linearly with size of proof
- Space and speed problem
Implicit LF ($\text{LF}_i$)

- Solution: Modify LF to reconstruct missing subterms while type checking
- Inherits all the advantages of LF
- $\text{LF}_i$ typing rules are not directly useful for type checking or type inference
- Only use to reconstruct object and corresponding LF typing derivation
Proof Generation

- PCC infrastructure so far is simple, easy to trust and automatic
- All the difficult tasks have been delegated to code and proof producers
  - Generate code annotations
  - Prove verification conditions
• Sometimes generation of annotations and of proofs can be automated

• For memory safety, any memory-safe high-level language can be used

• Policy is guaranteed to hold by design of static and run-time checks

• High-level type checker acts as theorem prover
Interaction between untrusted PCC tools (continuous lines) and trusted PCC infrastructure (interrupted lines)
PCC Beyond Types

• So far, focused on typed-based security properties

• Can change type system by changing type rules, with no change to infrastructure

• Can also enforce more complex safety properties than are usually associated with types
A (familiar) privacy policy
• Privacy safety policy can be implemented as a precondition on send function

• Need way to reflect history of functions

• Could keep track of state internally

• Instead, modify VCGen and symbolic evaluator to keep track of the public/private state
Histories \( H \ ::= \ x \mid \text{event } V \ H \)

Events \( V \ ::= \ \text{init} \mid \text{read} \mid \text{send} \)

Formulas \( F \ ::= \ldots \mid \text{publicState } H \mid \text{privateState } H \)

\[
\text{publicState (event init } H) \quad \text{(INIT)}
\]

\[
\text{publicState } H \quad \text{(SEND)}
\]

\[
\text{publicState (event send } H) \quad \text{(READ)}
\]

**VCGen Extensions**
\[
SE(i, \sigma) = \begin{cases} 
\ldots \\
SE(i + 1, \sigma[r_H \leftarrow (\sigma (\text{event read } r_H))]) & \text{if } \Pi_i = \text{call read} \\
\text{publicState}(\sigma r_H) \wedge \\
SE(i + 1, \sigma[r_H \leftarrow (\sigma (\text{event send } r_H))]) & \text{if } \Pi_i = \text{call send}
\end{cases}
\]

Symbolic Evaluator Extensions
• $r_H$ register can be used in loop invariants and function pre/postconditions

• Could be extended for general purpose function calls

• Any safety policy that could be enforced by an interpreter using run-time checking could in principle be enforced by PCC

• But at compile time!
Conclusion
Improvements

- Operates at **load-time** before agent code is installed in the host system
- Trusted computing base is **small**
- PCC operates on agents in **native-code** form
- PCC is **general**
- **Complements** cryptographic authentication
Drawbacks

• Difficult to produce code annotations and proofs
  • In general, a human is required
• PCC shifts this burden from the code receiver to the code producer
  • Computational power
• Knowledge of why code obeys safety
Review

• Author explained concept well
• Very readable
• Surprised that JVM, TAL basically implement the same concept
• Liked discussion of alternatives
Thank you.

He ain’t heavy...
He’s my proof.

Questions?