Proof Carrying Code

Dennis Kafura
Advantages

♦ General, flexible
  – able to express variety of safety properties
  – accommodates variety of languages

♦ Low-risk, automatic –
  – minimal safety-critical infrastructure
  – infrastructure easily automated

♦ Efficient
  – efficient proof-checking process
  – no consumer modification
  – no consumer monitoring

♦ No trust required
  – consumer need not know/trust the identity of the code producer
  – consumer need not know/trust the process by which the code was produced
PCC Architectures - Elements

Code Producer

Annotated executable (AE)

Proof Producer

Prove SP

Safety proof (PRF)

Untrusted

Trusted

VCGen

Scan AE and generate safety predicate

Safety predicate (SP)

Verify that PRF proves SP

Install & Run

Code Consumer
Architecture – Low Latency

Code Producer

- Annotated executable (AE)
  - Scan AE and generate safety predicate
  - Safety predicate (SP)

Code Consumer

Proof Producer

- Prove SP
  - Safety proof (PRF)
  - Verify that PRF proves SP

Untrusted

Trusted

Install & Run
Architecture – Large Proof

Code Producer

Annotated executable (AE)

Proof Producer

Prove SP

Safety proof (PRF)

Untrusted

Trusted

Install & Run

Code Consumer

Scan AE and generate safety predicate

Verify that PRF proves SP

Safety predicate (SP)

VCGen
Architecture – Separate Execution Platform

Code Producer

Annotated executable (AE)

Proof Producer

Prove SP

Safety proof (PRF)

Untrusted

Trusted

Code Consumer

VCGen

Scan AE and generate safety predicate

Safety predicate (SP)

Verify that PRF proves SP

Install & Run
VCGen

Example Annotations:
- Loop invariants
- Call target

Pre/post-conditions:
- Untrusted code
- Exported functions

- Includes configuration data (pre/post conditions)
- Example safety conditions:
  - Initial and first loop invariant checks
  - Call target address matches annotation
Example

```c
int main(ENTRY tab[], int len, ACCESS acc) {
    for(j=0; j<len; j++) {
        if(tab[j].access <= acc) {int p = tab[j].price; ... }
    }
}
```
Example: pre/post condition

\[ Pre_{\text{main}} = \forall i. 0 \leq i \land i < \text{len} \supset \text{entry}(\text{mem}, \text{tab} + i \times 16, \text{acc}) \]

\[ Post_{\text{main}} = \text{true} \]

where:

\[ \text{entry}(m, e, a) = \text{saferd}(m, e + 0) \land \text{saferd}(m, e + 4) \land \text{saferd}(m, e + 8) \land \text{sel}(m, e + 0) \leq a \supset \text{saferd}(m, e + 12) \]

The predicate saferd \((e_1, e_2)\) is valid if in the memory state denoted by \(e_1\) it is safe to read from the address denoted by \(e_2\).
Intermediate Language

Variables \( Vars \) \( x \)

Variable sets \( P(Vars) \) \( s \)

Labels \( Label \) \( l \)

Expressions \( Expr \) \( e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid \)
\( \text{sel} (e_1, e_2) \mid \text{upd} (e_1, e_2, e_3) \)

Predicates \( Pred \) \( P ::= \text{true} \mid \text{false} \mid P_1 \land P_2 \mid P_1 \lor P_2 \mid \forall x. P_x \mid \)
\( e_1 = e_2 \mid e_1 \not= e_2 \mid e_1 \geq e_2 \mid e_1 < e_2 \mid \)
\( \text{saferd} (e_1, e_2) \mid \text{saewr} (e_1, e_2, e_3) \)

Instructions \( Instr \) \( c ::= \text{SET} \ x, e \mid \text{ASSERT} \ P \mid \text{CALL} \ l \mid \text{RET} \mid \)
\( \text{BRANCH} \ P_1 \rightarrow l_1 \mid P_2 \rightarrow l_2 \mid \)
\( \text{INV} \ P, s \mid \text{MEMRD} \ e \mid \text{MEMWR} \ e_1, e_2 \)

\( \text{sel} (e_1, e_2) \) : the contents of memory address \( e_2 \) in memory state \( e_1 \)

\( \text{upd} (e_1, e_2, e_3) \) : the new memory state obtained from the state \( e_1 \) by updating the location \( e_2 \) with the value denoted by \( e_3 \).

\( \text{INV} P, s \) : requires that predicate \( P \) be valid at the corresponding program point and declares the maximal set of variables that might be modified on all loop paths that contain this instruction.
Intermediate Language for Example

```c
int main(ENTRY tab[], int len, ACCESS acc) {
    for(j=0; j<len; j++) {
        if(tab[j].access <= acc) {int p = tab[j].price; ... }
    }
}
```

```
SET   j, 0
Loop: INV  j ≥ 0, {j}
       BRANCH j < len → L1 □ j ≥ len → Done
L1:    BRANCH sel(mem, tab + 16 × j) ≤ acc → L2 □ ... < acc → Next
L2:    MEMRD tab + 16 × j + 12
       SET   p, sel(mem, tab + 16 × j + 12)
       ...  
Next: SET   j, j + 1  
       BRANCH true → Loop
```
Symbolic Evaluator

The symbolic evaluator computes the result (the safety predicate) as a symbolic expression by a linear pass through the code (because all loops are required to have an invariant)

\[
SE(i + 1, \rho[x \leftarrow \rho(e)], \mathcal{L})
\]
\[
\rho(P) \land SE(i + 1, \rho, \mathcal{L})
\]
\[
(\rho(P_1) \supset SE(i_1, \rho, \mathcal{L})) \land (\rho(P_2) \supset SE(i_2, \rho, \mathcal{L}))
\]

\[
\text{safe rd} (\rho(m), \rho(e)) \land SE(i + 1, \rho, \mathcal{L})
\]
\[
\text{safe wr} (\rho(m), \rho(e_1), \rho(e_2)) \land SE(i + 1, \rho, \mathcal{L})
\]
\[
\rho(P) \land \forall y_1 \ldots y_k. \rho'(P) \supset SE(i + 1, \rho', \mathcal{L}[i \leftarrow \rho'])
\]

if \( IL_i = \text{SET } x, e \)
if \( IL_i = \text{ASSERT } P \)
if \( IL_i = \text{BRANCH } P \rightarrow i_1 \rightarrow i_2 \)
\[
i_1 < i \supset IL_{i_1} = \text{INV}
\]
\[
i_2 < i \supset IL_{i_2} = \text{INV}
\]
if \( IL_i = \text{MEMRD } e \)
if \( IL_i = \text{MEMWR } e_1, e_2 \)
if \( IL_i = \text{INV } P, \{x_1, \ldots, x_k\} \) and \( i \notin \text{Dom}(\mathcal{L}) \)
\[
\{y_1, \ldots, y_k\} \text{ are new variables}
\]
\[
\rho' = \rho[x_1 \leftarrow y_1, \ldots, x_k \leftarrow y_k]
\]
if \( IL_i = \text{INV } P, s \) and \( i \in \text{Dom}(\mathcal{L}) \)
if \( IL_i = \text{CALL } l \)
\[
\Pi_i = (Pre, Post, \{x_1, \ldots, x_k\})
\]
\[
\{y_1, \ldots, y_k\} \text{ are new variables}
\]
\[
\rho' = \rho[x_1 \leftarrow y_1, \ldots, x_k \leftarrow y_k]
\]
if \( IL_i = \text{RET} \) and \( \Pi_f = (Pre, Post, s) \)
\[
\rho(Pre) \land \forall y_1 \ldots y_k. \rho'(Post) \supset SE(i + 1, \rho', \mathcal{L})
\]
\[
\rho(Pre) \land \check{\text{Eq}}(\rho, \mathcal{L}_i, s)
\]
\[
\rho(Pre) \land \forall y_1 \ldots y_k. \rho'(Post) \supset SE(i + 1, \rho', \mathcal{L})
\]
\[
\rho(Pre) \land \check{\text{Eq}}(\rho, \mathcal{L}_f, s)
\]
Safety Predicate

The safety predicate of a function is obtained by evaluating it symbolically starting in a state that maps the global variables and function formal parameters to new variables over which the predicate is universally quantified.

```c
int main(EXIT tab[], int len, ACCESS acc) {
    for(j=0; j<len; j++) {
        if(tab[j].access <= acc) {int p = tab[j].price; ... }
    }
}
```

\[
\forall mem. \forall tab. \forall len. \forall acc. \\
(\forall i. 0 \leq i \land i < len \supset entry(mem, tab + 16 \times i, acc) \supset \\
(0 \geq 0 \land \\
\forall j. (j \geq 0 \supset \\
(j < len \supset ((sel(mem, tab + 16 \times j) \leq acc \supset \\
\text{safefd}(mem, tab + 16 \times j + 12)) \land \\
(sel(mem, tab + 16 \times j) > acc \supset j + 1 \geq 0)) \land \\
(j \geq len \supset true)))
\]
Checking and Generating Proof

Checking proofs:

Generating proofs:

• Since safety predicates are expressed using first-order logic, the proof generator must be a theorem prover for a fragment of first-order logic. In some cases, an interactive theorem prover may be needed.

• The theorem prover must not only be able to prove safety predicates but must be also capable of generating detached proof of them expressed in the particular logic used by the code consumer.