Proof-Carrying Code

COS 598E

Kedar Swadi

Can Software Be Trusted?

- Untested software
- Mobile Code
- Active Networks
- Software Updates

[Appel 02]

What Should Software Guarantee?

- Accesses its own memory
- Accesses variables properly
- Limited resource usage
  - Holds limited number of locks
  - Runs for limited number of steps

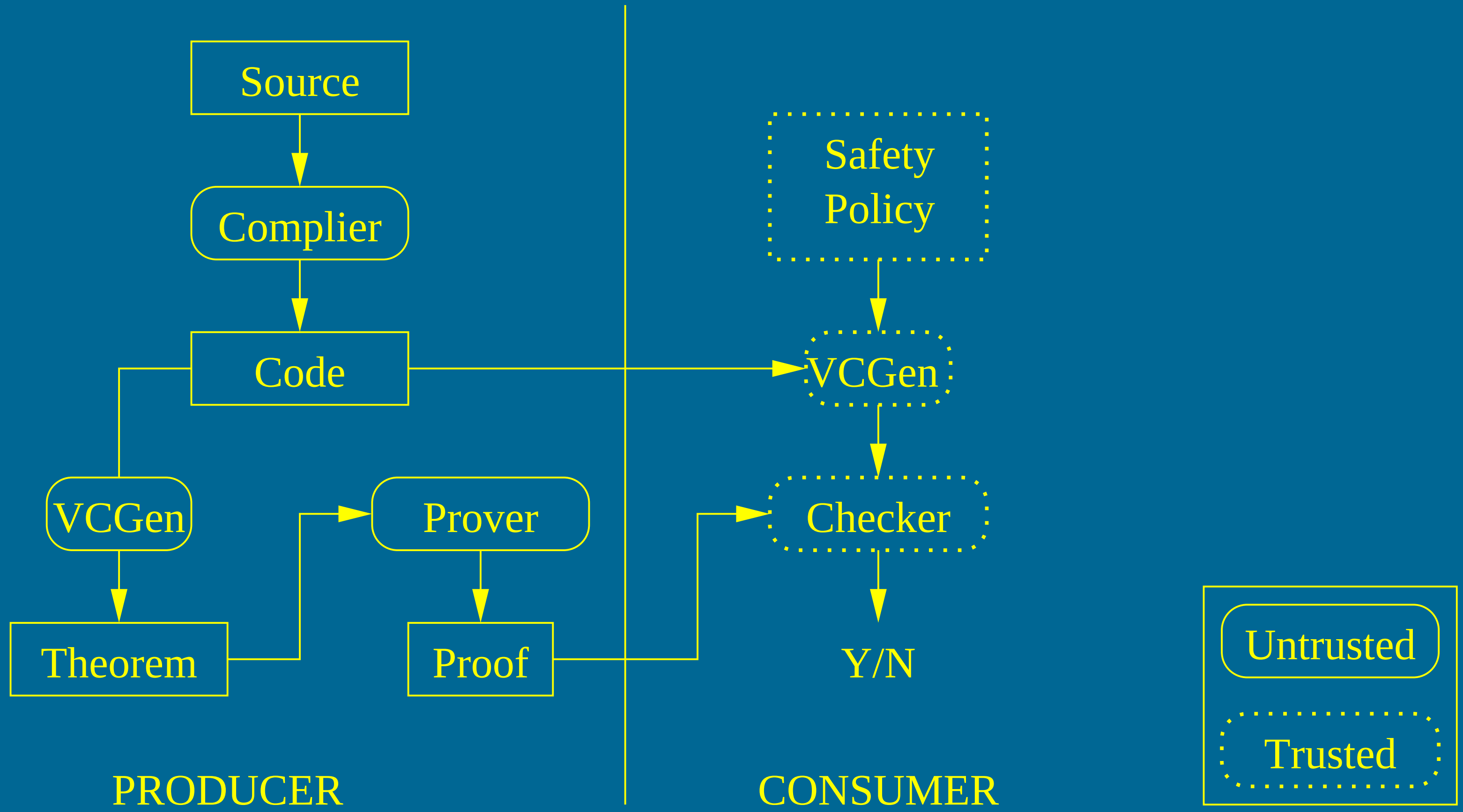
How Can We Trust Software?

- Trust an authority
- Hardware-based techniques
- Software fault isolation
- Prove it safe!!

Proving Software Safe

- Producer and consumer agree on what "safety" constitutes
- Producer sends code + proof of conformance to safety
- Consumer checks for the validity of proof before execution

Proof-Carrying Code Framework



Verification Condition Generator

- Verification Condition = Safety Theorem
- Uses assembly code and safety invariants at code points
- Based on Hoare-logic style pre(post)condition generator
- *Symbolic Evaluator*

An Example Language

Instr	::=	`add r1, r2, r3` | `load r1 r2 c` | `mov r1 r2`
		| `store r1 r2 c` | `beq r1, c, L` | `jump L`
		| `halt`
Types τ	::=	`int` | `array (int)`
Inv σ	::=	$\cdot \mid \{V : \tau\} \mid \sigma_1 \wedge \sigma_2 \mid V$ op $c \mid \sigma_1 \supset \sigma_2$
Value V	::=	$r_i \mid r_i + c \mid m[V]$
op	::=	$< \mid > \mid \leq \mid \geq \mid = \mid \neq$

Symbolic Evaluator(SE)

Precondition	Postcondition
$\sigma_1 = SE(I_1, \sigma_2)$	σ_1
I_1	I_1
$\sigma_2 = SE(I_2, \sigma_3)$	$\sigma_2 = SE'(I_1, \sigma_1)$
$\vdots$	$\vdots$
$\sigma_{n-1} = SE(I_n, \sigma_n)$	$\sigma_{n-1} = SE'(I_{n-1}, \sigma_{n-2})$
I_n	I_n
σ_n	$\sigma_n = SE'(I_n, \sigma_{n-1})$

- SE can either compute the *weakest* precondition, or the *strongest* postcondition
- Either approach is equivalent

Symbolic Evaluator(SE)

Instr I	$SE(I)$
add r1 r2 r3	$\lambda\sigma.[r2 + r3/r1]\sigma$
load r1 r2 c	$\lambda\sigma.[m[r2 + c]/r1]\sigma$
store r1 r2 c	$\lambda\sigma.[r2/m[r1 + c]]\sigma$
mov r1 r2	$\lambda\sigma.[r2/r1]\sigma$
beq r1 c L	$\lambda\sigma.r1 = c \supset \sigma_L \wedge r1 \neq c \supset \sigma$
jump L	$\lambda\sigma.\sigma_L$
halt	$\lambda\sigma.\ \cdot$

Theorem of Program Safety

- Safe to execute next instruction
- Safe to jump to any branch target, T, from any of its sources, S_T
- Postconditions of jumps at least as strong as preconditions of jump targets.

Theorem of Program Safety II

F: I_F

$\vdots$

jump G

G: I_G

$\vdots$

jump H

H: I_H

$\vdots$

halt

$$\forall I_{f \in \{F,G,H\}}.\ I_f \supset \sigma_f$$

Producing Theorems From Programs

```
L1:   {r1 : array(int), r2 : int}
      {m[r1 + r2 + 0] : int}
      add r3 r2 r1
      {m[r3 + 0] : int}
      load r2 r3 0
      {r2 : int}
      jump L2
      ⋮
L2:   {r2 : int}
```

Safety Theorem = $r1 : array(int) \wedge r2 : int \supset m[r1 + r2 + 0] : int$

$\bigwedge \quad \ldots$

Producing Proofs For Theorems

Inference Rules	Typing Rules
$\dfrac{\Gamma \vdash P_1 \;\wedge\; P_2}{\Gamma \vdash P1} \wedge E1$	$\dfrac{\Gamma \vdash ra : int}{\Gamma \vdash ra + c : int}\ C\_ADD$
$\dfrac{\Gamma \vdash P_1 \;\wedge\; P_2}{\Gamma \vdash P2} \wedge E2$	$\dfrac{\Gamma \vdash V : int \quad \Gamma \vdash ra : int}{\Gamma \vdash ra + V : int}\ V\_ADD$
$\dfrac{P1, \Gamma \vdash P2}{\Gamma \vdash P_1 \;\supset\; P_2} \supset I$	$\dfrac{\Gamma \vdash ra : array(int) \quad \Gamma \vdash V : int}{\Gamma \vdash ra + V : array(int)}\ V\_ARRAY$
$\dfrac{}{P1, \Gamma \vdash P_1}\ CTX$	$\dfrac{\Gamma \vdash V : array(int)}{\Gamma \vdash m[V] : int}\ M\_ARRAY$
$\vdots$	

Producing Proofs For Theorems II

$$I_1 = r1 : array(int) \wedge r2 : int$$

Safety Theorem = $I_1 \supset m[r1 + r2 + 0] : int$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{I_1 \vdash I_1}\,{\scriptstyle CTX}}{I_1 \vdash r1 : array(int)}\,{\scriptstyle \wedge E1} \qquad \dfrac{\dfrac{\dfrac{}{I_1 \vdash I_1}\,{\scriptstyle CTX}}{I_1 \vdash r2 : int}\,{\scriptstyle \wedge E2}}{I_1 \vdash r2 + 0 : int}\,{\scriptstyle C\_ADD}}{I_1 \vdash r1 + r2 + 0 : array(int)}\,{\scriptstyle V\_ARRAY}}{\dfrac{I_1 \vdash m[r1 + r2 + 0] : int}{I_1 \supset m[r1 + r2 + 0] : int}\,{\scriptstyle \supset I}}\,{\scriptstyle M\_ARRAY}$$

Producing Proofs For Theorems II

$$I_1 = r1 : array(int) \wedge r2 : array(int)$$

Safety Theorem = $I_1 \supset m[r1 + r2 + 0] : int$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{I_1 \vdash I_1}\,{\scriptstyle CTX}}{I_1 \vdash r1 : array(int)}\,{\scriptstyle \wedge E1} \qquad \dfrac{\dfrac{???}{I_1 \vdash r2 : int}}{I_1 \vdash r2 + 0 : int}\,{\scriptstyle C\_ADD}}{I_1 \vdash r1 + r2 + 0 : array(int)}\,{\scriptstyle V\_ARRAY}}{I_1 \vdash m[r1 + r2 + 0] : int}\,{\scriptstyle M\_ARRAY}}{I_1 \supset m[r1 + r2 + 0] : int}\,{\scriptstyle \supset I}$$

Using PCC for optimisations

PCC can deal with many optimisations

- Dead-code elimination
- Instruction scheduling
- **Array-bounds-check eliminations**

Producing Proofs For Optimisations

```
for(i=0;i<n;i++) {
  ⋮
                       if i > n goto error;
b = a[i];              if i < 0 goto error;
                       b = a[i] ;
  ⋮
}
```

$$\frac{ra : array(int, n) \quad 0 \leq V < n}{m[ra + V] : int}\ ARR$$

Producing Proofs For Optimisations II

Weakest Precondition Approach

```
        mov ri 0
loop:
```

$\{ri \geq 0 \;\;\wedge\;\; ra : array(int, n)\}$

$\{ri \geq n \supset \sigma_{error} \wedge ri < n \supset m[ra + ri] : int\}$

```
        bge ri n error
```

$\{m[ra + ri] : int\}$

$\vdots$

$\{m[ra + ri] : int\}$

```
        load rb ra, ri
```

$\{rb : int\} \;= \sigma_G$

```
        jump G
error :
```

Producing Proofs For Optimisations III

Strongest Postcondition Approach

```
        mov ri 0
loop:
```

$\{ri \geq 0, ri : int, ra : array(int, n)\}$

```
        bge ri n error
```

$\{ri \geq 0, ri < n, ri : int, ra : array(int, n)\}$

$\vdots$

$\{ri \geq 0, ri < n, ri : int, ra : array(int, n)\}$

```
        load rb ra, ri
```

$\{ri \geq 0, ri < n, ri : int, ra : array(int, n), rb : int\}$

```
        jump G
error :
```

What Could Go Wrong?

Adversary could change the input

- Code is modified
- Proof is modified
- Code and proof are modified

What do you have to trust?

- Encoding of the safety policy
- VCgen
- Proof checker

Trusting Type Safety As A Safety Policy

- Type safety encompasses many safety criteria
  - Instruction safety
  - System call safety
  - Function I/O behaviour
- Flexible and easy to configure
- Loop invariants can be automatically generated

What do you have to trust?

- Encoding of the safety policy
- VCgen
- Proof checker

Trusting the VCGen

- VCGen (23,000 lines of C for Cedilla)
- Machine-instruction semantics
- Encoding of typing rules
- Derivation of post-conditions through substitutions

What do you have to trust?

- Encoding of the safety policy
- VCgen
- Proof checker

Trusting the Proof Checker

- Parse logic and safety axioms
- Parse program proofs
- Handle basic arithmetic
- Should be as small and readable as possible

Foundational PCC

- Minimising the TCB
  - Minimise trust in instruction decoding
  - Removing trust from the type system
  - TCB only has logic/arithmetic/machine semantics
- More flexible

Instruction Decoding

Instructions syntax

3	i	j		k
26	21	16	5	0

Instruction Semantics

$$\begin{aligned} \mathrm{add}(\mathrm{i}, \mathrm{j}, \mathrm{k}) \;\; = & \\ & \lambda r, m, r', m'.\;\; r'(i) = r(j) + r(k) \\ & \qquad\qquad \wedge\; (\forall x \neq i.\, r'(x) = r(x)) \wedge\;\; m' = m \end{aligned}$$

Instruction Decoding II

- Use a well-factored syntax specification for architectures
- E.g. SLED specification
- Use higher-order logic to maintain factoring in encoding
- 1035+600 lines of “readable” Twelf

Foundational PCC

- Minimising the TCB
  - Minimise trust in instruction decoding
  - Removing trust from the type system
  - TCB only has logic/arithmetic/machine semantics
- More flexible

Semantic Model for Types

- Types are semantic judgements on machine values
- Types are not a part of the TCB
- New predicates on values allow defining new type system
- Flexibility of using different type systems for parts of code

Defining Types as Predicates

- $\tau \Rightarrow \lambda v.\ \tau(v)$
- char $\Rightarrow \lambda v.\ (v \geq 0 \wedge v < 256)$
- ref$(\tau) \Rightarrow \lambda v.\ (v \geq 255 \wedge m[v] : \tau)$
- $\tau_1 \cap \tau_2 \Rightarrow \lambda v.\ (\tau_1(v) \wedge \tau_2(v))$
- $v : \tau \Rightarrow pf(\tau(v))$

$$\frac{v : \tau_1 \qquad v : \tau_2}{v : \tau_1 \cap \tau_2} \cap I \quad \Rightarrow \quad \begin{array}{l} \cap I : pf(\tau_1(v)) \to pf(\tau_2(v)) \\ \qquad \to pf((\tau_1 \cap \tau_2)(v)). \end{array}$$

Relating Machine Steps to Types I

- Machine semantics relate one state to another
- State given by register bank and memory (r, m)
- Types are much more complex
  - readable, writable, allocated sets
  - various bookkeeping parameters

Relating Machine Steps to Types II

- `load` : $(r, m) \rightarrow (r', m')$
- $(r, m) \approx \Phi$
- *load* : $\Phi \rightarrow \Phi'$
- $(r', m') \approx \Phi'$

Advantages/Disadvantages of Semantic Models

- All type judgements must have a proof
- Type predicates are difficult to construct
  - Contravariant recursion
  - Mutable types
  - Higher-order kinds
- Types can be added without changing existing proofs

Practical Considerations

- Size of proofs
- Time to check proofs

Proof-Size Reduction Techniques

- Increasing the lemma base
  - Choose high-level lemmas for frequently occurring subproofs
  - Increases time to load lemma base
  - Difficult to find such a set

Proof-Size Reduction Techniques II

- Choosing a smart representation
  - Implicit representation : LF_i
  - $(P \supset (P \wedge P) \Rightarrow \texttt{imp\_i} * *(\texttt{and\_i} * * \texttt{p p})$
  - Varying gains $1/4$ to $1/100$ sizes of proofs
  - Saves time for checking

Proof-Size Reduction Techniques III

- Oracle-based checking approaches
  - Middle path between checking and proving
  - Proofs are hints to a non-deterministic checker
  - Use non-determinism where search space is small
  - Proof sizes less than 20% of code sizes on average

End

*