

# Proof-Carrying Code (and Other Proof-Carrying Things)

Lujo Bauer

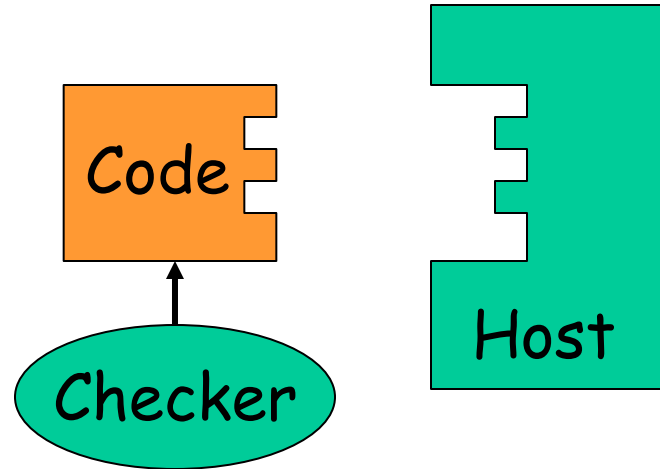
18-732

Fall 2010

# Motivation

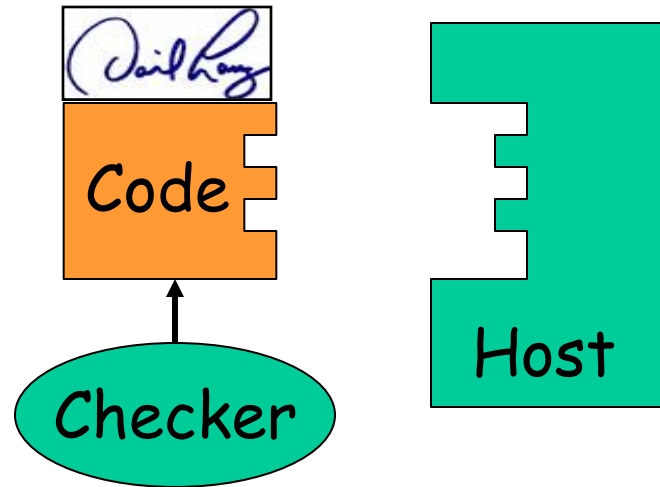
- Execution of untrusted code has become an integral part of everyday “Internet experience”
  - Document handlers and viewers
    - RealAudio, Flash player, Acrobat, Ghostview, ...
  - Downloaded code
    - Freeware, shareware, trialware,...
  - Mobile code
    - Applets, Javascript, ...
  - Games, peer-to-peer applications, ...
- Untrusted code given all of the privileges of the user running it
  - Risks due to faulty or malicious code are very high

# Extensibility and Security



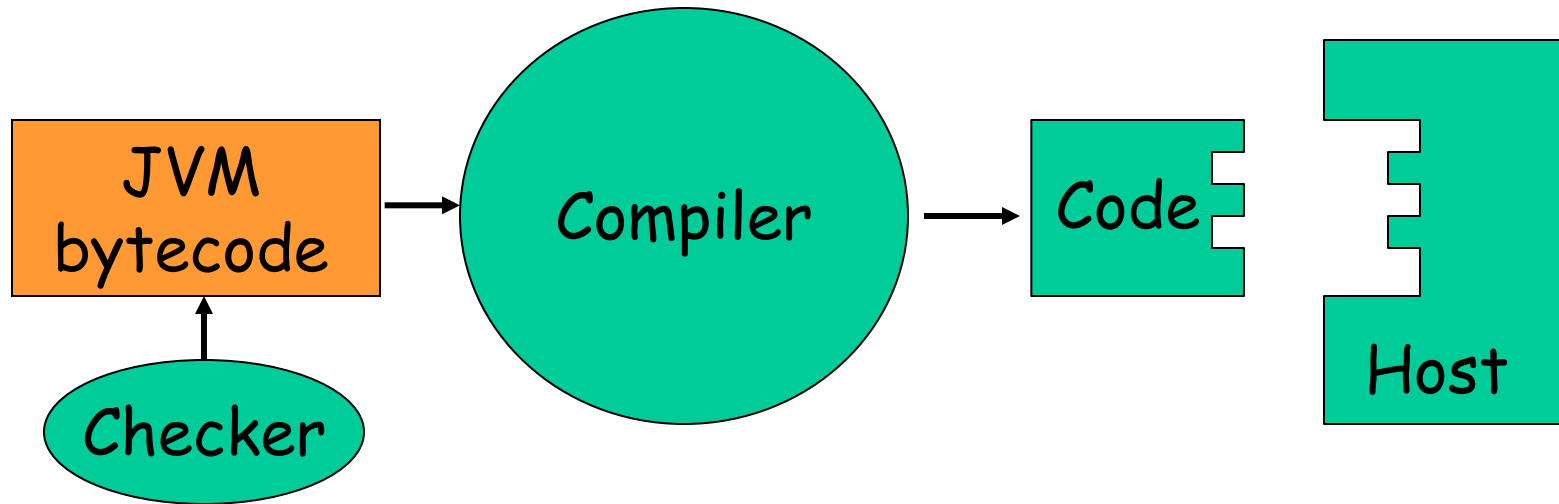
- Extensible systems are common
  - Device drivers, applets, servlets, components, ...
- Would like to have assurance of compatibility
  - To protect against malicious code producers
  - To protect against programming errors

# Assurance Support: Digital Signatures



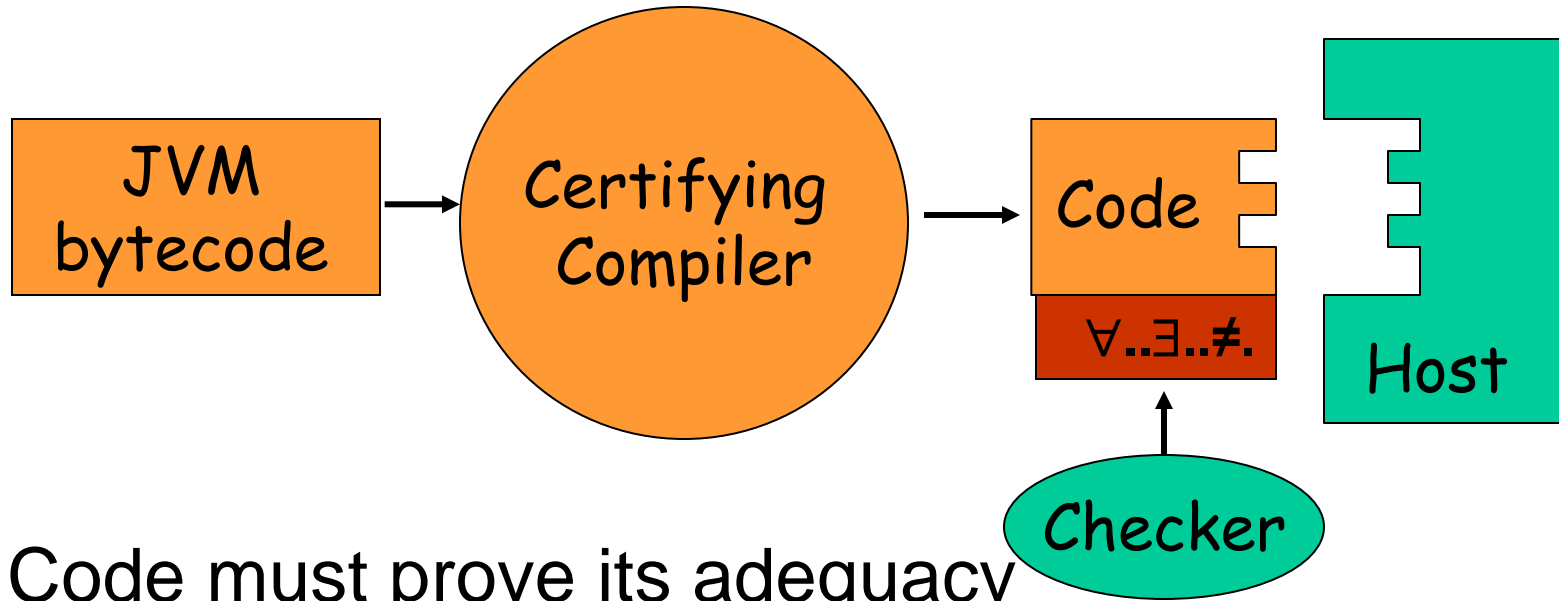
- Trust some code producers
- Not a behavioral assurance
- Does not scale well to many code producers

# Assurance Support: Java Bytecode



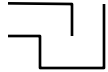


- Does not check what you run
- Must trust a complex compiler
- The compiler might not fit in the host system
- Checker works for only one safety policy
  - Java type safety

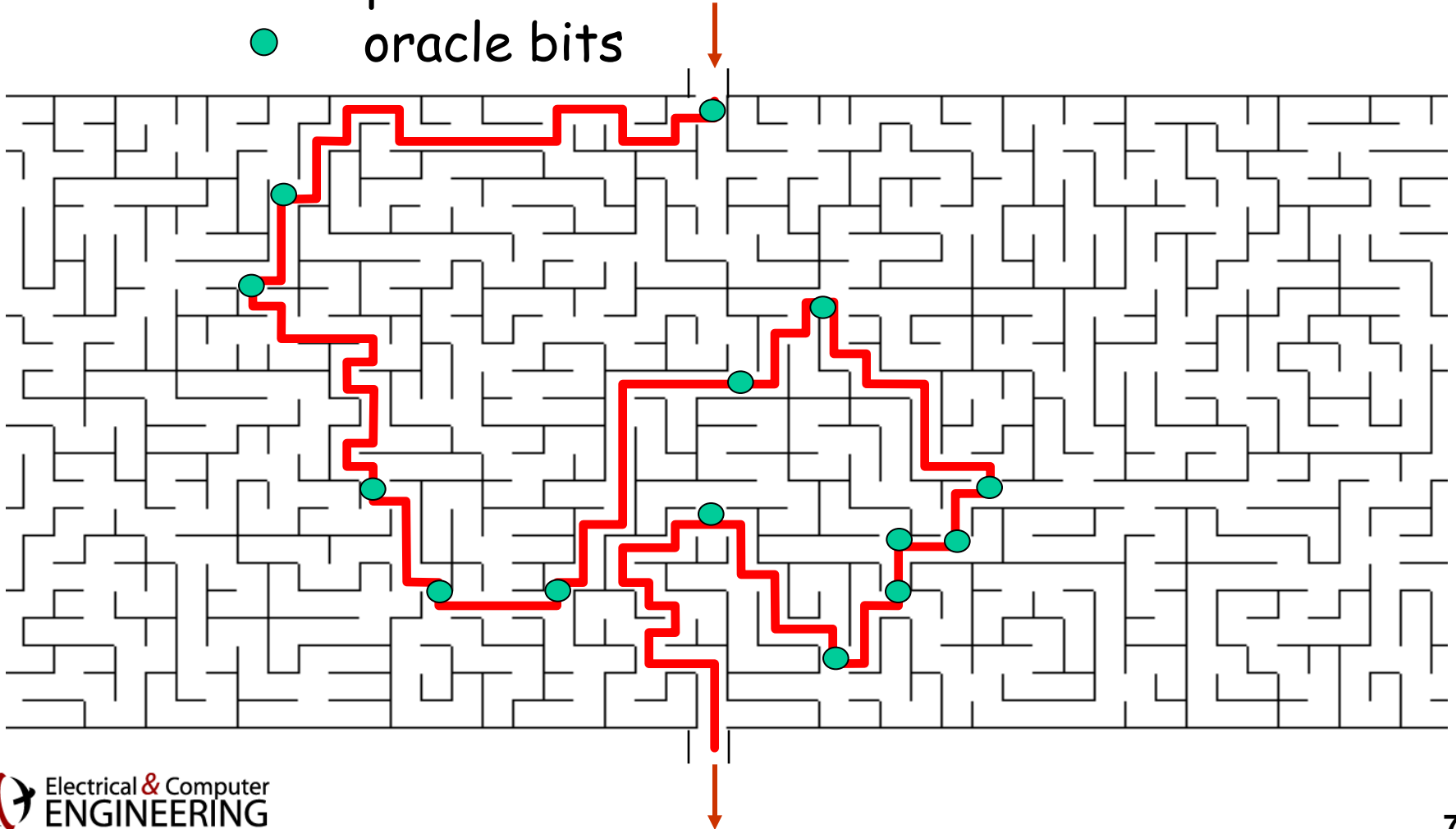
# Assurance Support: Proof-Carrying Code (PCC)



- Code must prove its adequacy
- Hard to prove but easy to check
- Works even for machine code—check what you run
- One (simple) checker for many policies
- Trust very little

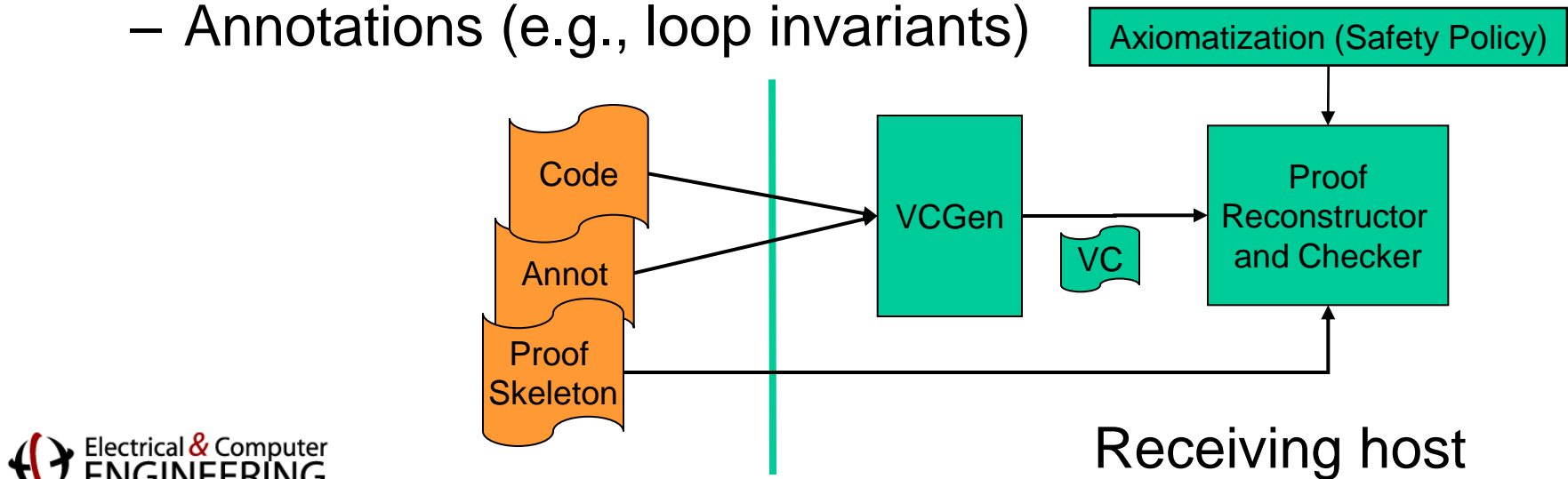
# Proof-Carrying Code: An Analogy

Legend:  code  
 proof  
 oracle bits



# One Instantiation of PCC

- Checker is composed of
  - A verification-condition generator (VCgen)
  - A proof reconstructor and checker
  - Parameterized by a description of the safety policy
- Proof is composed of
  - A skeleton of a derivation of the verification condition
  - Annotations (e.g., loop invariants)



# An Example: Type-Based Memory Safety

**PRE**  $a : \text{array}(\text{bool}, n)$

$r \leftarrow 1$

$i \leftarrow n$

$L_0$ : **INV** =  $r : \text{bool} \wedge i \leq n$ , **REG** =  $\{ m, a, n \}$

if  $i < 0$  goto  $L_1$

$t \leftarrow a + i$

$t \leftarrow *t$

$r \leftarrow r \& t$

$i \leftarrow i - 1$

goto  $L_0$

$L_1$ : return  $r$

**POST**  $r : \text{bool}$

Safety policy:

- Memory reads and writes are allowed between  $a$  and  $a + n$
- Only booleans can be written to  $a$
- If the function returns, it must return a boolean
- $0$  and  $1$  are the only booleans
- Expressed as preconditions and postconditions

# Safety Policy $\Rightarrow$ Axiomatization

$$\frac{A : \text{array}(T, L) \quad I \geq 0 \quad I \leq L}{\text{saferd}(A + I)} \text{rd} \qquad \frac{A : \text{array}(T, L) \quad I \geq 0 \quad I \leq L}{\text{sel}(M, A + I) : T} \text{typedrd}$$

$$\frac{A : \text{array}(T, L) \quad I \geq 0 \quad I \leq L \quad E : T}{\text{safewr}(A + I, E)} \text{wr}$$

$$\frac{}{0 : \text{bool}} \text{bool0}$$

$$\frac{}{1 : \text{bool}} \text{bool1}$$

$$\frac{}{E \leq E} \text{leqid}$$

$$\frac{I \leq E \quad I \geq 0}{I - 1 \leq E} \text{dec}$$

# Verification Condition Generation

Symbolic register file:

<b>a</b>	<b>a0</b>
<b>n</b>	<b>n0</b>
<b>m</b>	<b>m0</b>
<b>i</b>	<b>i0</b>
<b>r</b>	<b>r0</b>
<b>t</b>	<b>t0</b>

Assumptions:



PRE a : array(bool, n)

r ← 1

i ← n

L<sub>0</sub>: INV = r : bool ∧ i ≤ n, REG = { m, a, n }

if i < 0 goto L<sub>1</sub>

t ← a + i

t ← \*t

r ← r & t

i ← i - 1

goto L<sub>0</sub>

L<sub>1</sub>: return r

POST r : bool

# Verification Condition Generation

Symbolic register file:

<b>a</b>	<b>a0</b>
<b>n</b>	<b>n0</b>
<b>m</b>	<b>m0</b>
<b>i</b>	<b>i0</b>
<b>r</b>	<b>r0</b>
<b>t</b>	<b>t0</b>

Assumptions:

**a0 : array(bool, n0)**

→ **PRE** a : array(bool, n)

r ← 1

i ← n

L<sub>0</sub>: **INV** = r : bool ∧ i ≤ n, **REG** = { m, a, n }

if i < 0 goto L<sub>1</sub>

t ← a + i

t ← \*t

r ← r & t

i ← i - 1

goto L<sub>0</sub>

L<sub>1</sub>: return r

**POST** r : bool

# Verification Condition Generation

Symbolic register file:

<b>a</b>	<b>a0</b>
<b>n</b>	<b>n0</b>
<b>m</b>	<b>m0</b>
<b>i</b>	<b>i0</b>
<b>r</b>	<b>1</b>
<b>t</b>	<b>t0</b>

PRE a : array(bool, n)

→ r ← 1

i ← n

L<sub>0</sub>: INV = r : bool ∧ i ≤ n, REG = { m, a, n }

if i < 0 goto L<sub>1</sub>

t ← a + i

t ← \*t

r ← r & t

i ← i - 1

goto L<sub>0</sub>

L<sub>1</sub>: return r

POST r : bool

Assumptions:

a0 : array(bool, n0)

# Verification Condition Generation

Symbolic register file:

<b>a</b>	<b>a0</b>
<b>n</b>	<b>n0</b>
<b>m</b>	<b>m0</b>
<b>i</b>	<b>n0</b>
<b>r</b>	<b>1</b>
<b>t</b>	<b>t0</b>

PRE a : array(bool, n)

r ← 1

i ← n

→ L<sub>0</sub>: INV = r : bool ∧ i ≤ n, REG = { m, a, n }

if i < 0 goto L<sub>1</sub>

t ← a + i

t ← \*t

r ← r & t

i ← i - 1

goto L<sub>0</sub>

L<sub>1</sub>: return r

POST r : bool

Assumptions:

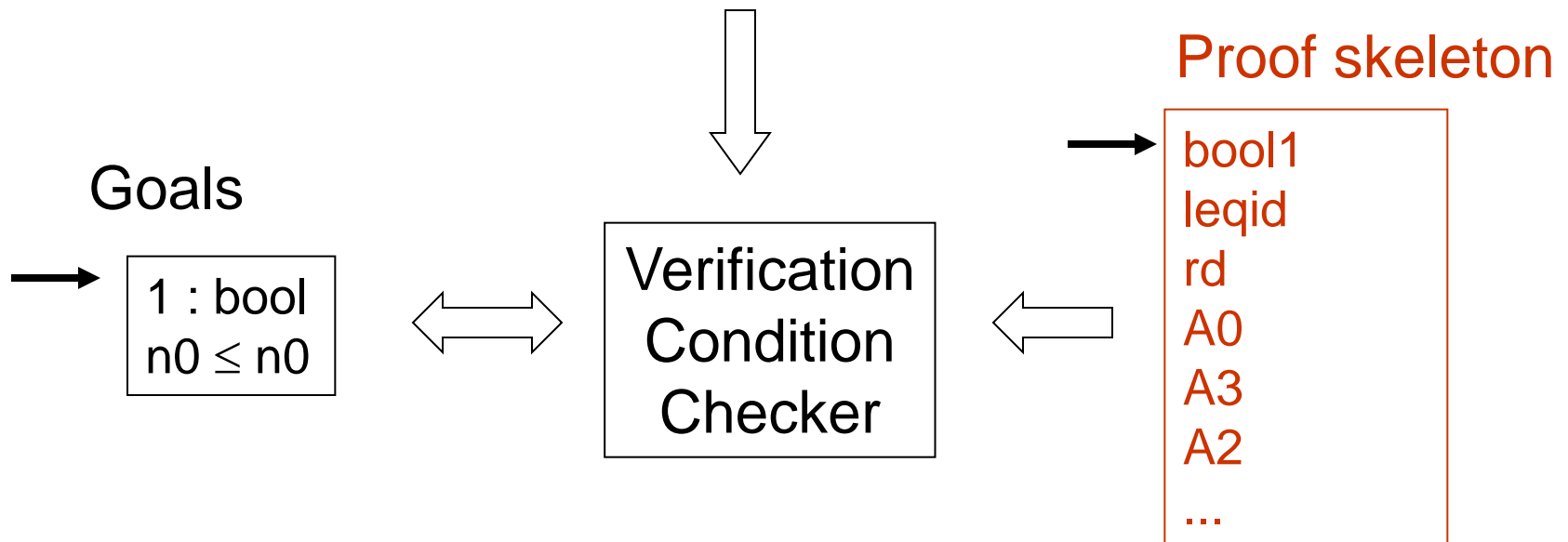
a0 : array(bool, n0)

Check: 1 : bool  
n0 ≤ n

# Checking Verification Conditions

Rules

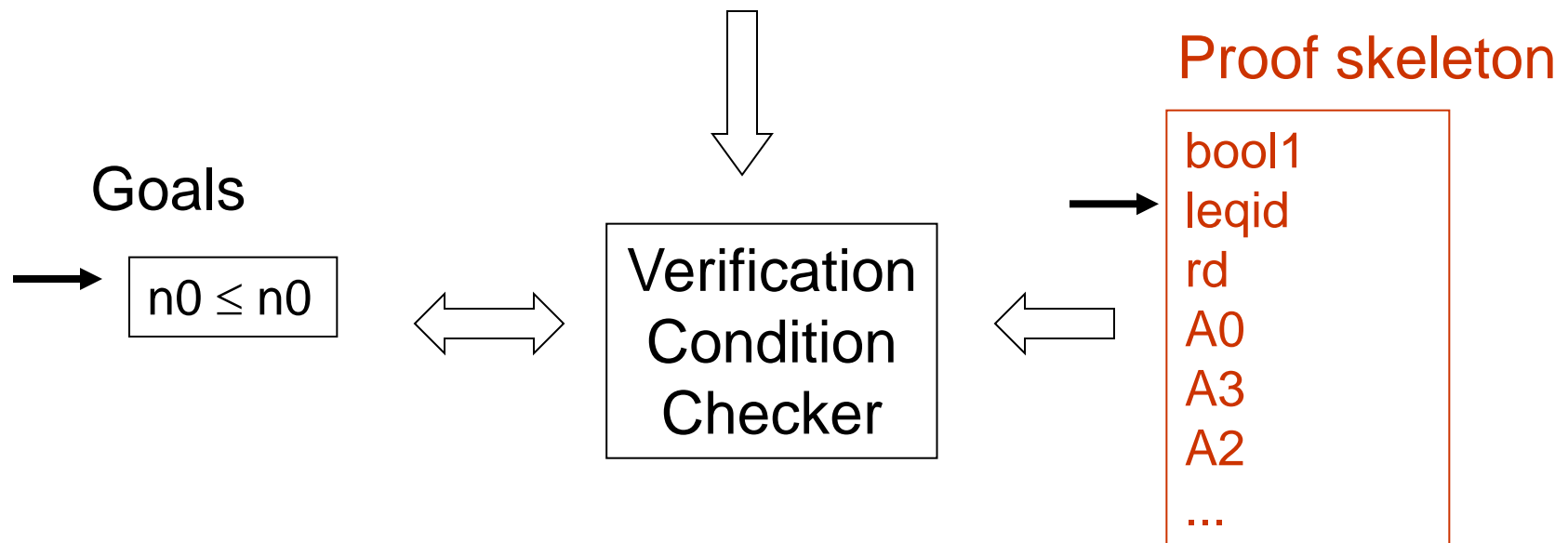
Safety policy rules, and  
A0:  $a0 : \text{array}(\text{bool}, n0)$



- The proof guides the VC checker in using the axiomatization and the assumptions to prove the goals

# Checking Verification Conditions

Rules Safety policy rules, and  
A0: a0 : array(bool, n0)



- Finished checking VCs
- Return to the VC generator and continue the scan

# Verification Condition Generation

Symbolic register file:

PRE  $a : \text{array}(\text{bool}, n)$

$r \leftarrow 1$

$i \leftarrow n$

$L_0$ :  $\text{INV} = r : \text{bool} \wedge i \leq n, \text{REG} = \{ m, a, n \}$

if  $i < 0$  goto  $L_1$

$t \leftarrow a + i$

$t \leftarrow *t$

$r \leftarrow r \& t$

$i \leftarrow i - 1$

goto  $L_0$

$L_1$ : return  $r$

POST  $r : \text{bool}$

a	a0
n	n0
m	m0
i	i1
r	r1
t	t1

Assumptions:

$a0 : \text{array}(\text{bool}, n0)$

$r1 : \text{bool}$

$i1 \leq n0$

# Verification Condition Generation

Symbolic register file:

<b>a</b>	<b>a0</b>
<b>n</b>	<b>n0</b>
<b>m</b>	<b>m0</b>
<b>i</b>	<b>i1</b>
<b>r</b>	<b>r1</b>
<b>t</b>	<b>t1</b>

PRE a : array(bool, n)

r ← 1

i ← n

L<sub>0</sub>: INV = r : bool ∧ i ≤ n, REG = { m, a, n }

→ if i < 0 goto L<sub>1</sub>

t ← a + i

t ← \*t

r ← r & t

i ← i - 1

goto L<sub>0</sub>

L<sub>1</sub>: return r

POST r : bool

Assumptions:

a0 : array(bool, n0)

r1 : bool

i1 ≤ n0

i1 ≥ 0

# Verification Condition Generation

Symbolic register file:

<b>a</b>	<b>a0</b>
<b>n</b>	<b>n0</b>
<b>m</b>	<b>m0</b>
<b>i</b>	<b>i1</b>
<b>r</b>	<b>r1</b>
<b>t</b>	<b>a0 + i1</b>

PRE a : array(bool, n)

r ← 1

i ← n

L<sub>0</sub>: INV = r : bool ∧ i ≤ n, REG = { m, a, n }

if i < 0 goto L<sub>1</sub>

→ t ← a + i

t ← \*t

r ← r & t

i ← i - 1

goto L<sub>0</sub>

L<sub>1</sub>: return r

POST r : bool

Assumptions:

a0 : array(bool, n0)

r1 : bool

i1 ≤ n0

i1 ≥ 0

# Verification Condition Generation

Symbolic register file:

<b>a</b>	<b>a0</b>
<b>n</b>	<b>n0</b>
<b>m</b>	<b>m0</b>
<b>i</b>	<b>i1</b>
<b>r</b>	<b>r1</b>
<b>t</b>	<b>a0 + i1</b>

PRE a : array(bool, n)

r ← 1

i ← n

L<sub>0</sub>: INV= r : bool ∧ i ≤ n, REG = { m, a, n }

if i < 0 goto L<sub>1</sub>

→ t ← a + i

t ← \*t

r ← r & t

i ← i - 1

goto L<sub>0</sub>

L<sub>1</sub>: return r

POST r : bool

Assumptions:

a0 : array(bool, n0)

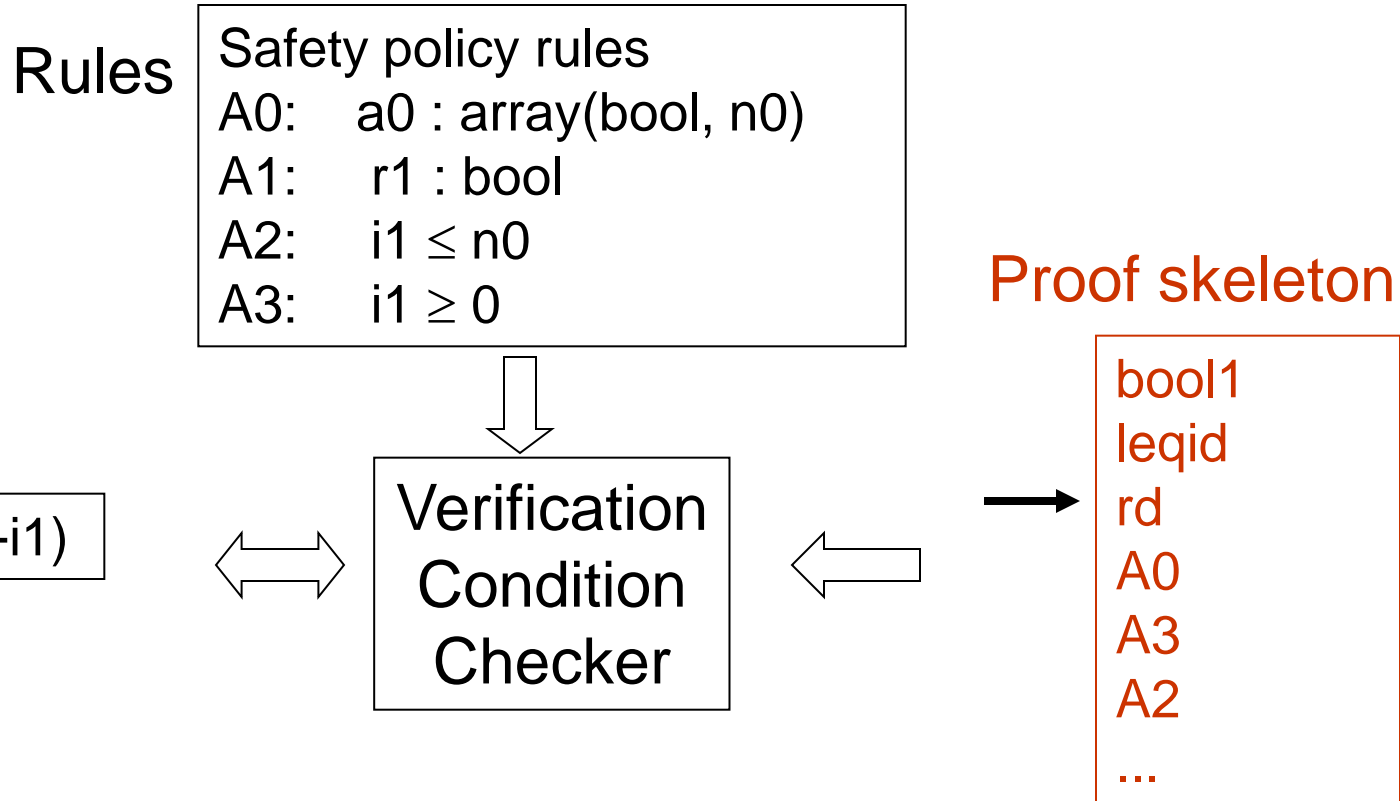
r1 : bool

i1 ≤ n0

i1 ≥ 0

**Check: saferd(a0+i1)**

# Checking Verification Conditions



- Three subgoals are produced

# Checking Verification Conditions

Rules

Safety policy rules

A0:  $a0 : \text{array}(\text{bool}, n0)$

A1:  $r1 : \text{bool}$

A2:  $i1 \leq n0$

A3:  $i1 \geq 0$

Goals

$a0 : \text{array}(T, L)$   
 $i1 \geq 0$   
 $i1 \leq L$

Verification  
 Condition  
 Checker

Proof skeleton

$\text{bool1}$   
 $\text{leqid}$   
 $\text{rd}$   
 $A0$   
 $A3$   
 $A2$   
 ...

# Checking Verification Conditions

Rules

Safety policy rules

A0: a0 : array(bool, n0)

A1: r1 : bool

A2:  $i1 \leq n0$

A3:  $i1 \geq 0$

Goals

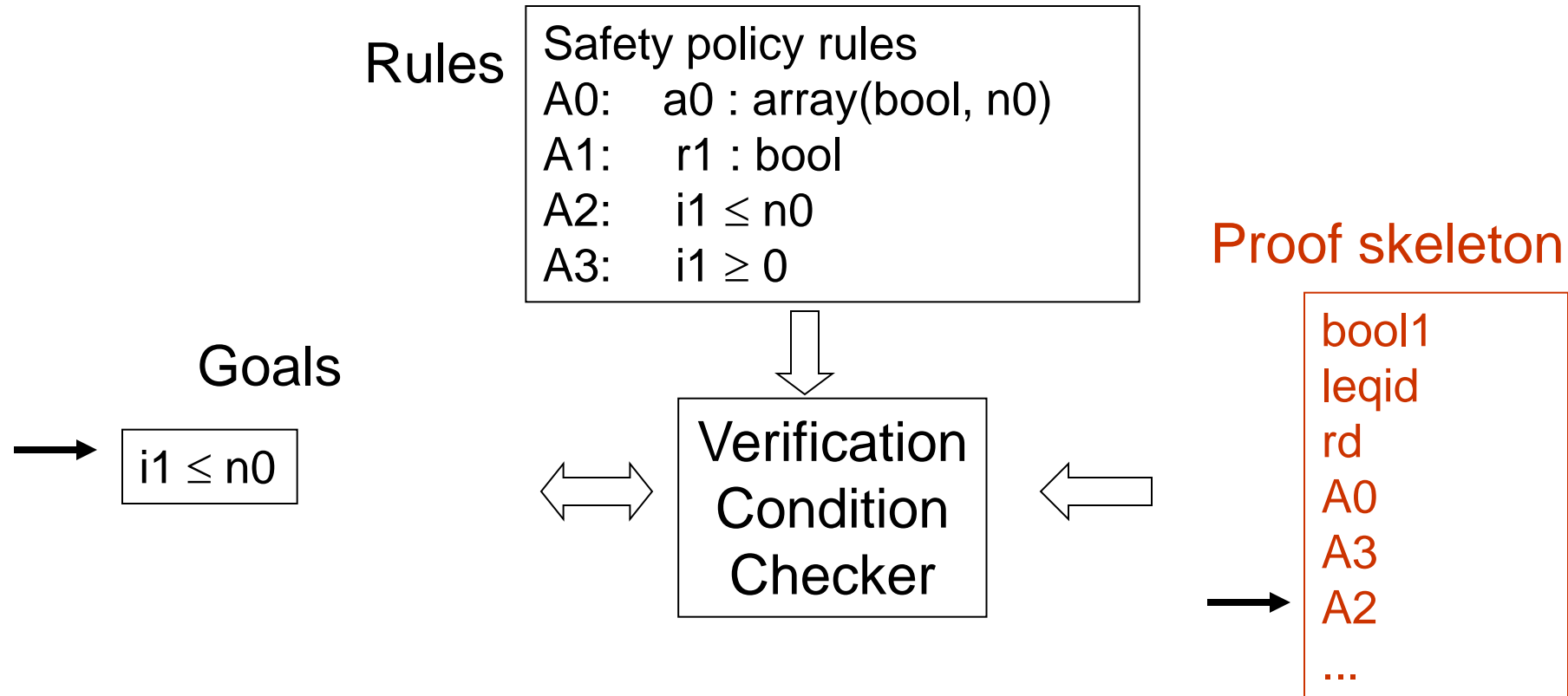
$i1 \geq 0$   
 $i1 \leq n0$

Verification  
Condition  
Checker

Proof skeleton

bool1  
leqid  
rd  
A0  
A3  
A2  
...

# Checking Verification Conditions



- Finished checking, return to VCGen, and so on

# Oracle-Based Checking

- VC Checker is a logic interpreter
  - The logic program is the axiomatization
  - The verification conditions are the goals
  - The proof is the oracle that guides the interpreter
  - No backtracking, simple and fast, policy independent
- The oracle of safety for our example is:
  - bool1, leqid, rd, A0, A3, A2, booland, A1, typerd, A0, A3, A2, dec, A2, A3, A1

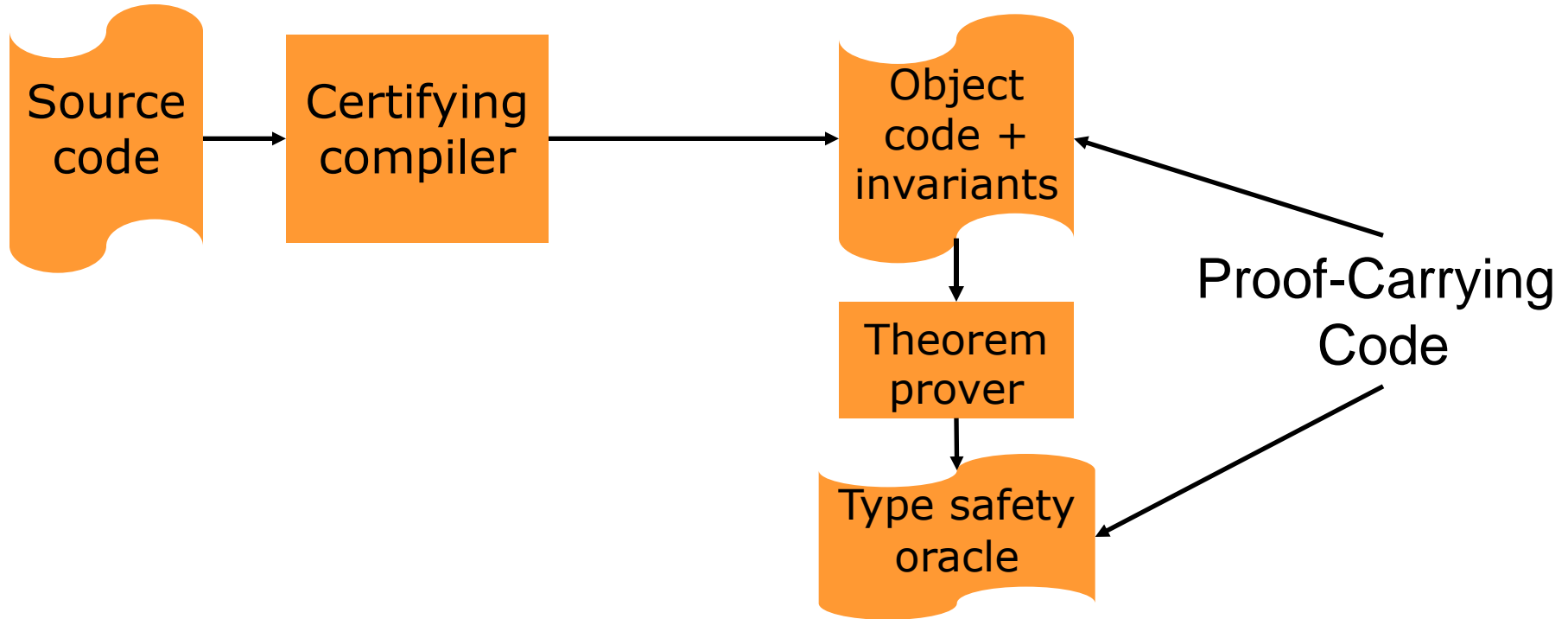
# A Logic-Programming Optimization

- Make the VC Checker smarter
  - Predict (conservatively) the usable rules at each step
  - The oracle must only select among those choices
    - For each step need  $\log_2(\text{nr\_choices})$  bits
  - The harder the proof/policy the more help is needed
- Any off-the-shelf logic programming optimization aimed at reducing backtracking will help

# PCC Automation

- PCC challenges:
  - Generating the loop invariants
  - Generating the proofs
  
- Automation idea (for type safety):
  - Start with a type-safe language
  - The proof exists at source level; just preserve it!
  - Compile the proof while compiling the code

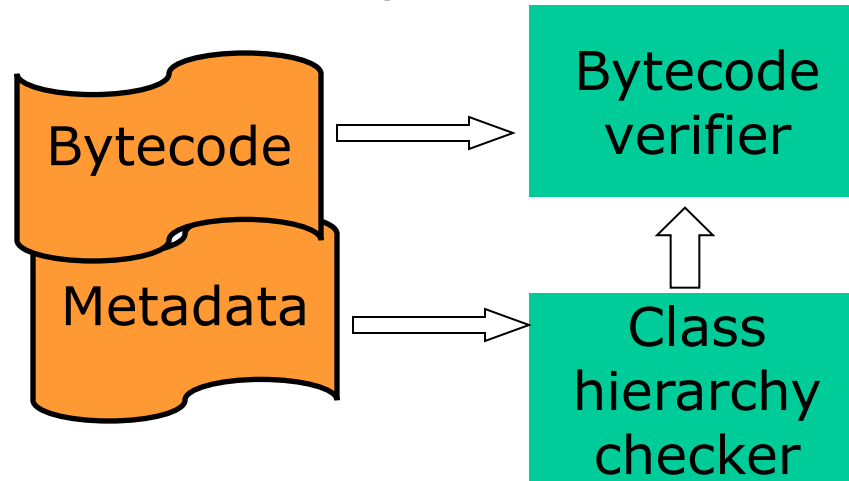
# Automation via Certifying Compilation



- Implementation:
  - The compiler emits typing invariants for registers
  - The prover reconstructs the proof of type safety

# Java Bytecode Verification

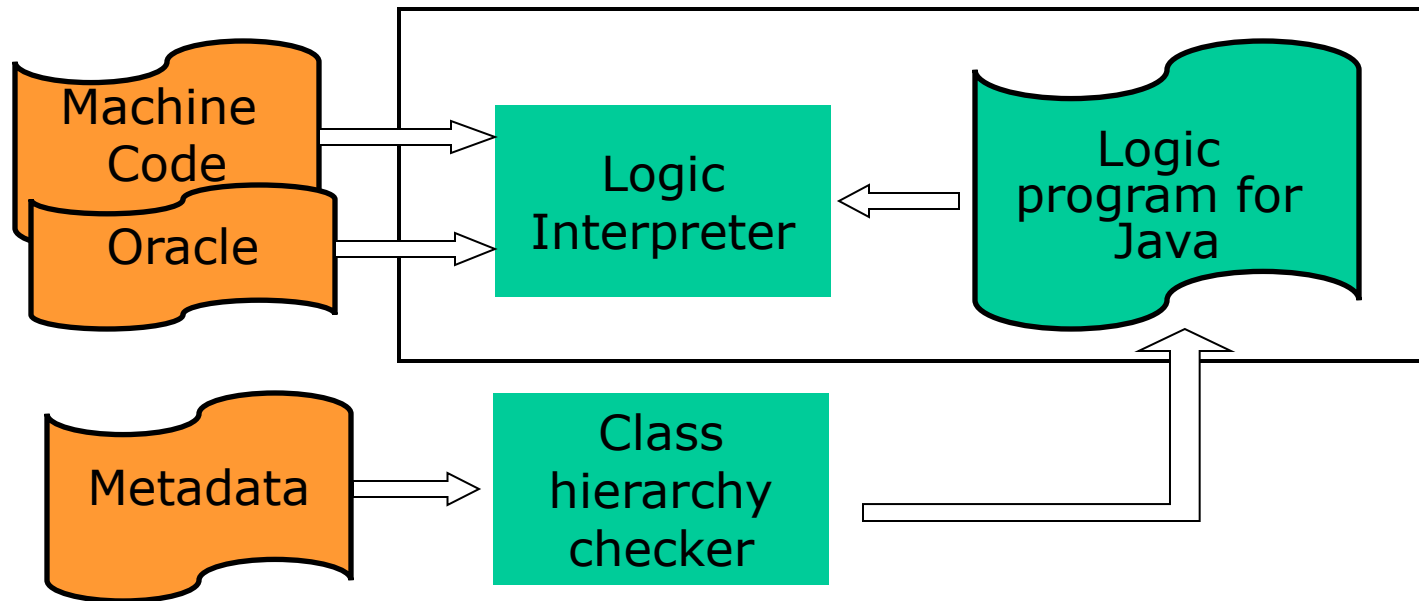
- A Java .class file contains
  - Code (bytecode format)
  - Metadata (describing the class hierarchy)



- Standard JVM verifier checks both
  - Uses the metadata to learn how to check the code

# Instantiating PCC to Java

- The PCC verifier for Java:



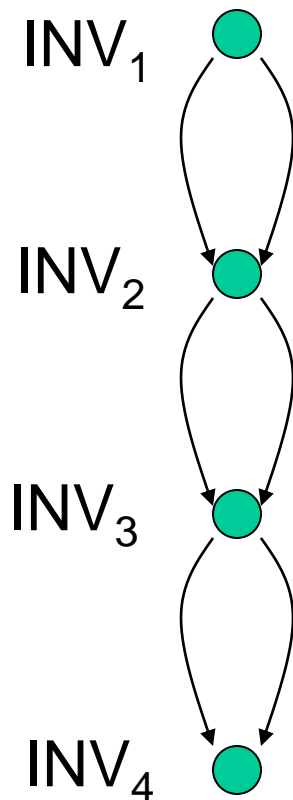
- Replace the bytecode verifier with PCC checker
- The metadata checker adds new logic rules to describe the class hierarchy

# The Touchstone Compiler

- Certifying compiler for Java bytecode to x86
  - Collaboration with Cedilla Systems in Pittsburgh
- Mostly a conventional optimizing compiler
  - Implemented in OCaml (50K lines)
- Emits type declarations for live variables at joins
- Emits hints for VCGen:
  - Loop structure of the program
  - Indirect function calls
  - Exception handling code

# Avoiding the Exponential Explosion

- VCGen tries to check every path through the code
  - Necessary for supporting arbitrary correctness proofs
  - But there could be an exponential number of paths



- Compiler places an invariant at each join
- Checking is done from invariant to invariant
- Linear checking in the size of code
- Sufficient for type checking

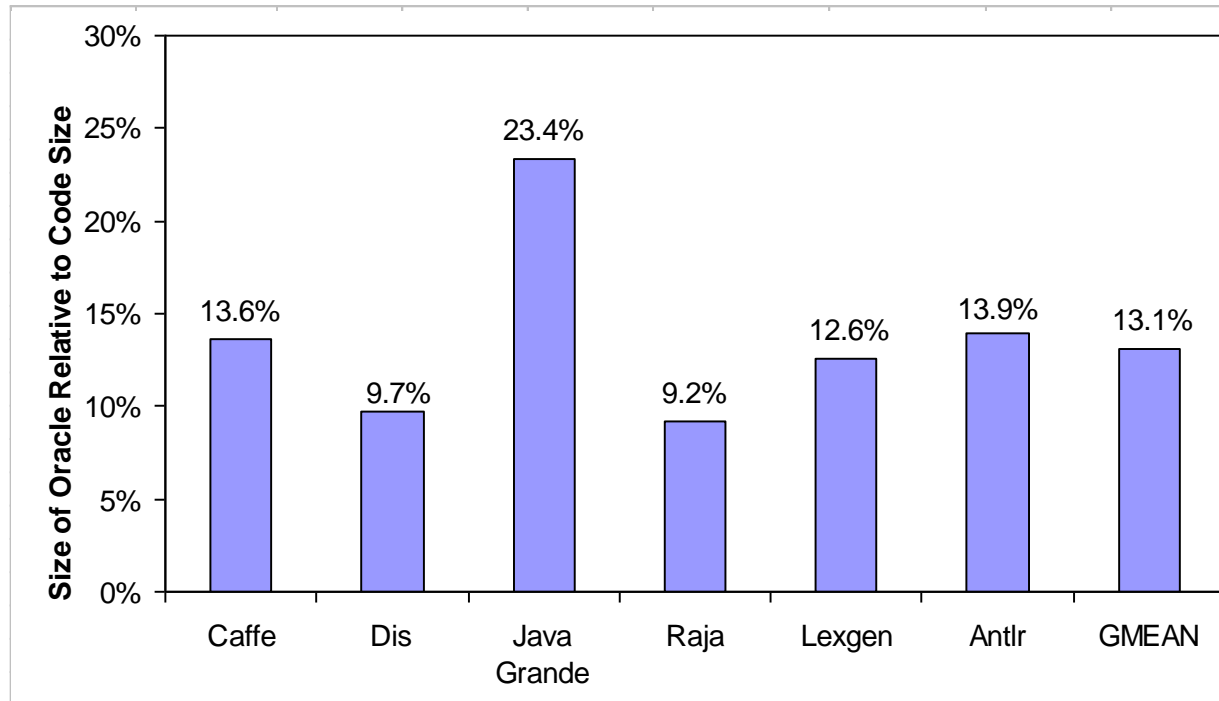
# The Kettle Theorem Prover

- Constructs the oracle
- Same structure as the proof checker
  - Except it does not use an oracle to select clauses
  - It must search and backtrack to find the proof
  - Then constructs an oracle describing the path taken
- Kettle is a conventional logic interpreter
  - Extended with a mechanism to incorporate decision procedures written in OCaml
  - Implemented in OCaml (7K lines)

# Experimental Results

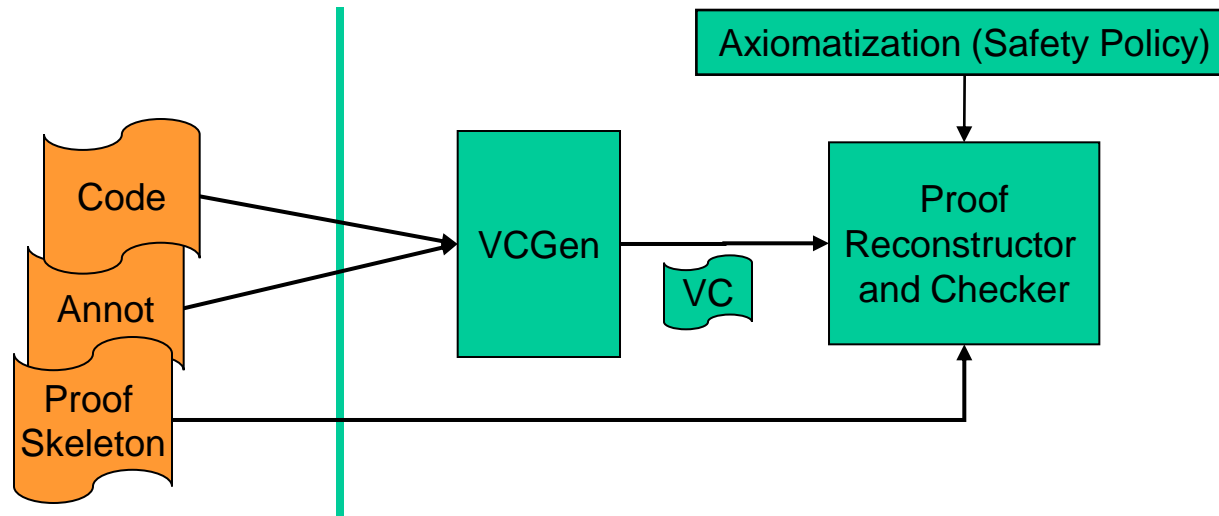
- Compiled several large Java programs
  - Kopi Java Development Suite (~ 80K lines of Java)
  - StarOffice (~ 140K lines of Java)
  - Sun's HotJava browser (~ 150K lines of Java)
- An extremely effective way to debug the compiler
  - A proof failure always points to a bug in the compiler
  - Found bugs where testing did not (e.g. exception handling)

# Oracle-Size vs Code Size



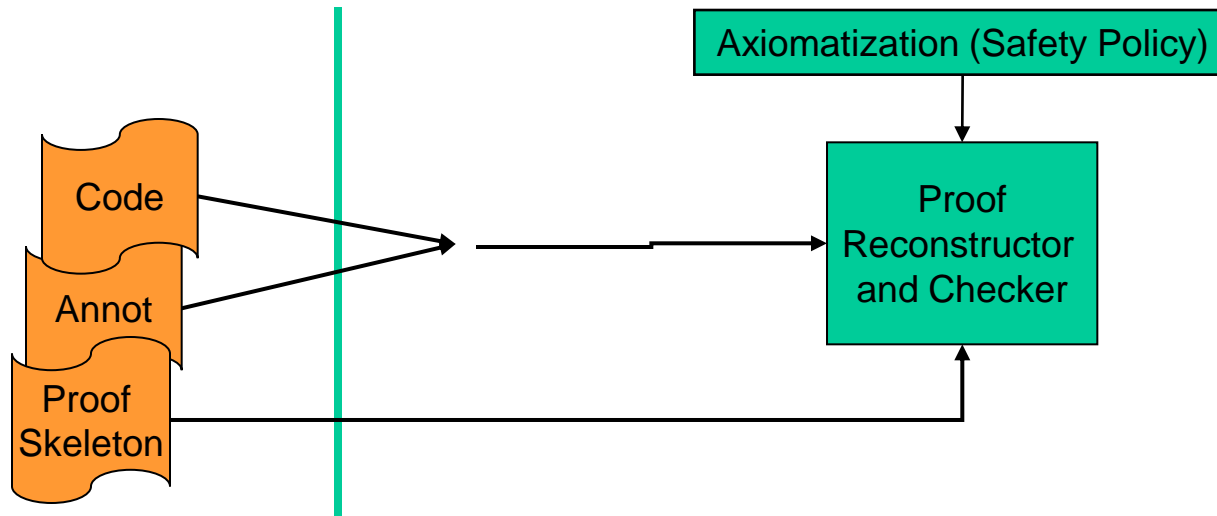
- Oracles are about 5-25% of the machine code
  - This is about 30 times smaller than  $LF_i$  proof skeletons
  - Invariants are now an issue (25% of the machine code)
- Oracle checking speed is about 12K/second

# Weaknesses?



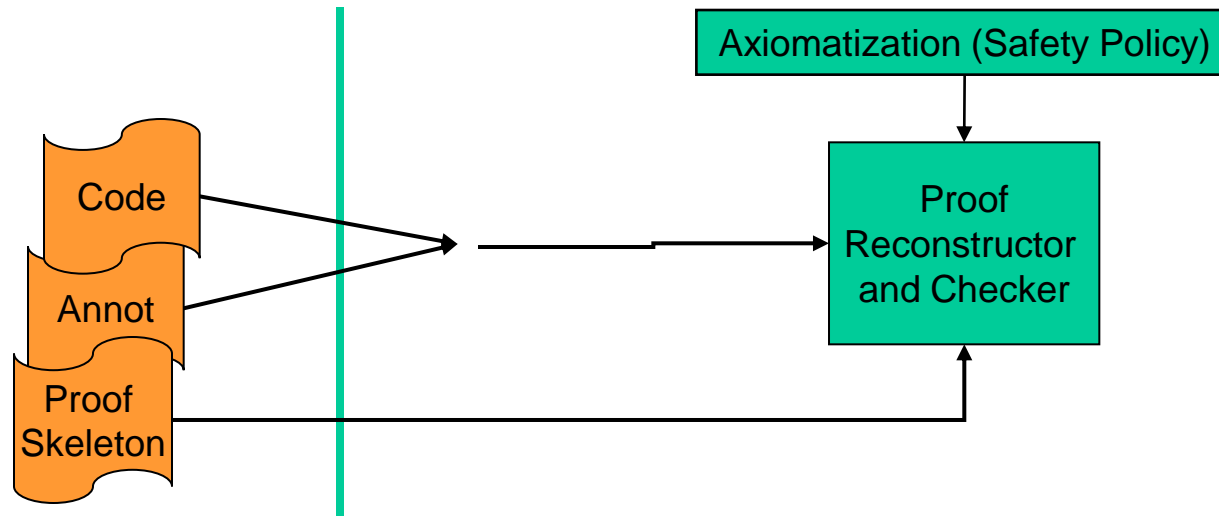
- Trusted computing base (TCB) could be smaller
  - VCGen + Axiomatization + Logic interpreter
- VCGen and axiomatization not reusable
  - Specific to particular programming language (Java)

# Foundational Proof-carrying Code



- No VCGen
- Axiomatization = machine instruction semantics + low-level safety policy
- Annotations = full type information
- Proof = witness of typecheckability

# Foundational Proof-carrying Code



- Advantage: smaller, more general TCB
  - No VCGen, no language-specific axiomatization
- Advantage: truly portable checker
  - Completely independent of programming language
- Disadvantage: larger proofs
  - All proofs built on machine instruction semantics – very low level

# What Can We Check with PCC ?

- Any property with a sound formalization
  - *If you can prove it, PCC can check it!*
- One proof-checker for many such policies!
  - Small commitment for open-ended extensibility
- Example: policies defined by safe interpreters
  - E.g., security monitors, memory safety, resource usage limits, ...
- ... can be enforced statically with PCC
  - Prove that all run-time checks would succeed
  - No run-time penalty

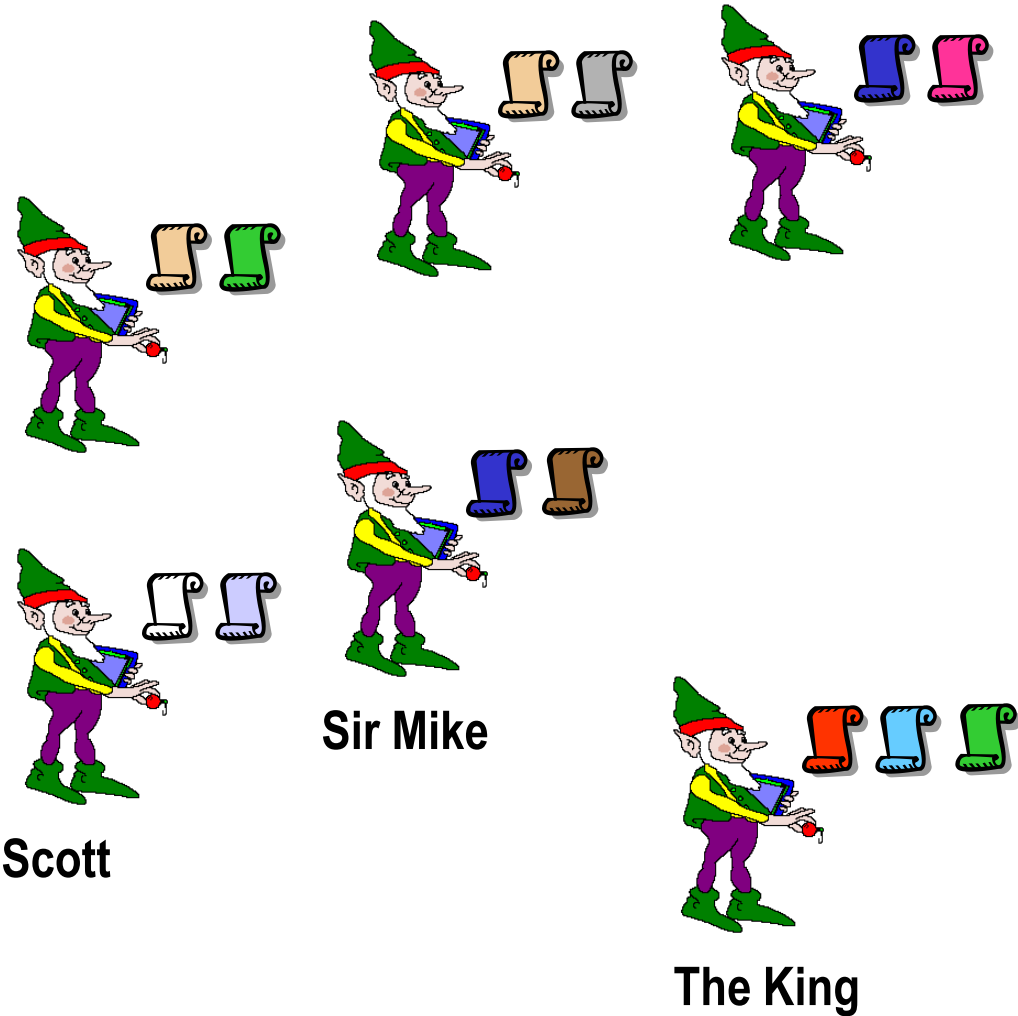
# PCC Summary

- Software must be executable and checkable
  - Sandboxes are not always appropriate
- Powerful safety checkers are kept simple by allowing them to consult proofs/oracles
  - A large TCB cannot be trusted
- One instance of PCC:
  - Checker = non-deterministic logic program interpreter
  - Oracle = resolves non-deterministic clause choices
  - Uses off-the-self logic programming technology
- PCC is automatic and practical for type safety
- More research is needed before we can push PCC beyond type safety

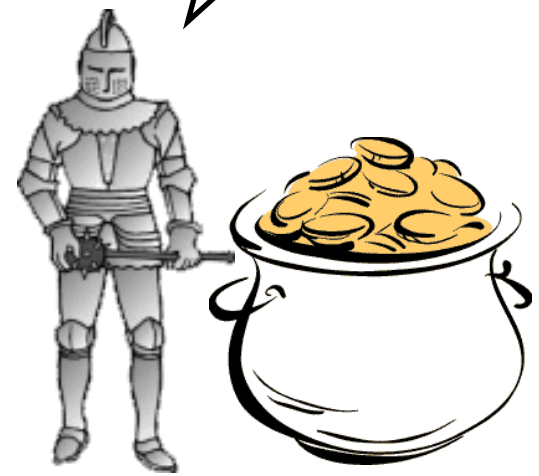
# Applying the Proof-Carrying Paradigm

- Another application: Proof-Carrying Authorization (PCA)  
[Appel and Felten, 1999]

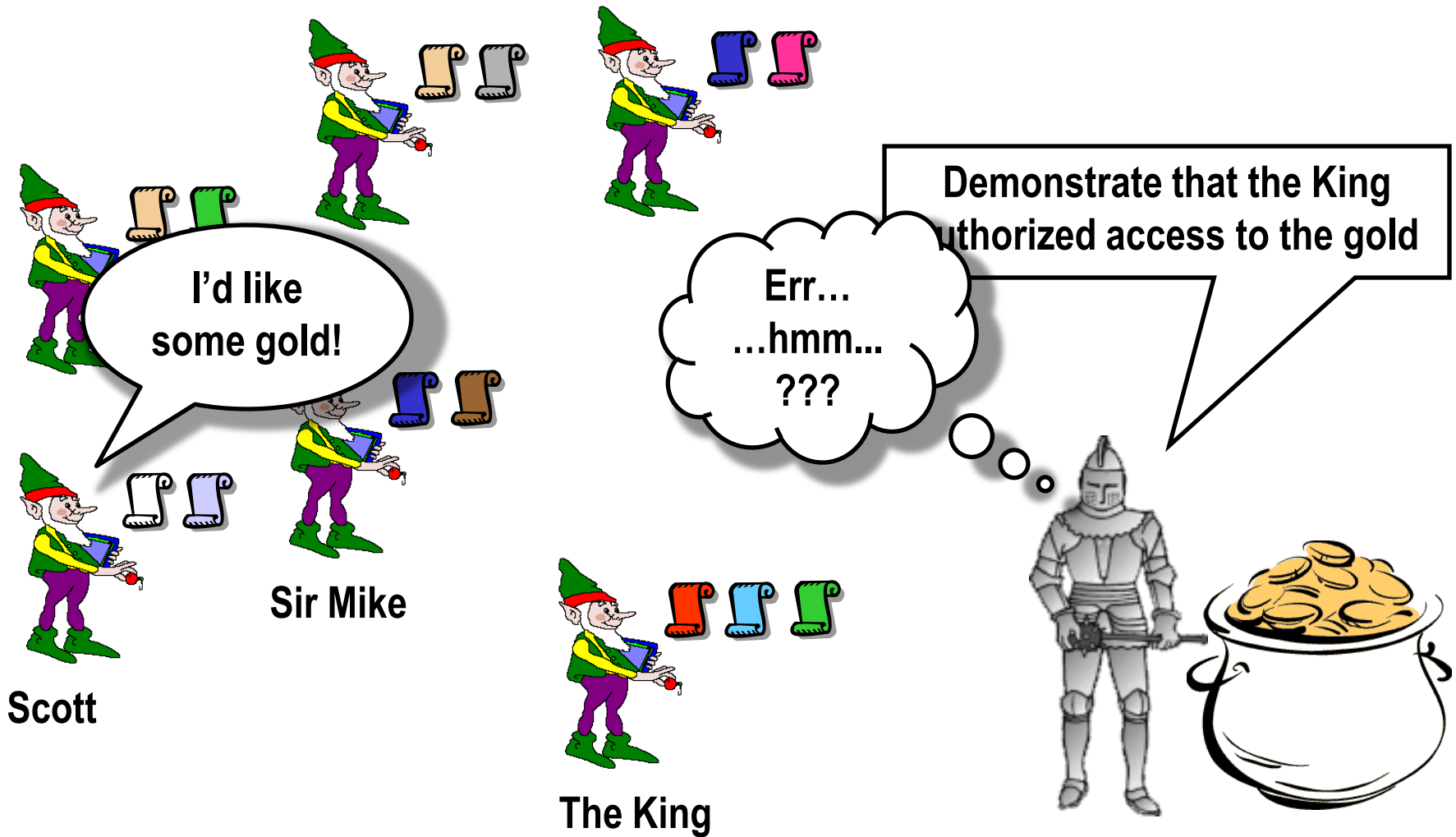
# A Distributed Authorization Scenario



Demonstrate that the King authorized access to the gold



# A Distributed Authorization Scenario



# Sample Access-Control Logic

Expressing beliefs:

- Alice says F
  - It can be inferred that Alice believes that F is true
- Alice signed F
  - Alice states (cryptographically) that she believes that F is true

Types of beliefs:

- Alice says open(resource, nonce)
  - Alice wishes to access a resource
- Alice says (Bob speaksfor Alice)
  - Alice wishes to delegate all authority to Bob
- Alice says delegate(Alice, Bob, door1)
  - Alice wishes to delegate authority over a specific resource to Bob

# Sample Logic Inference Rules

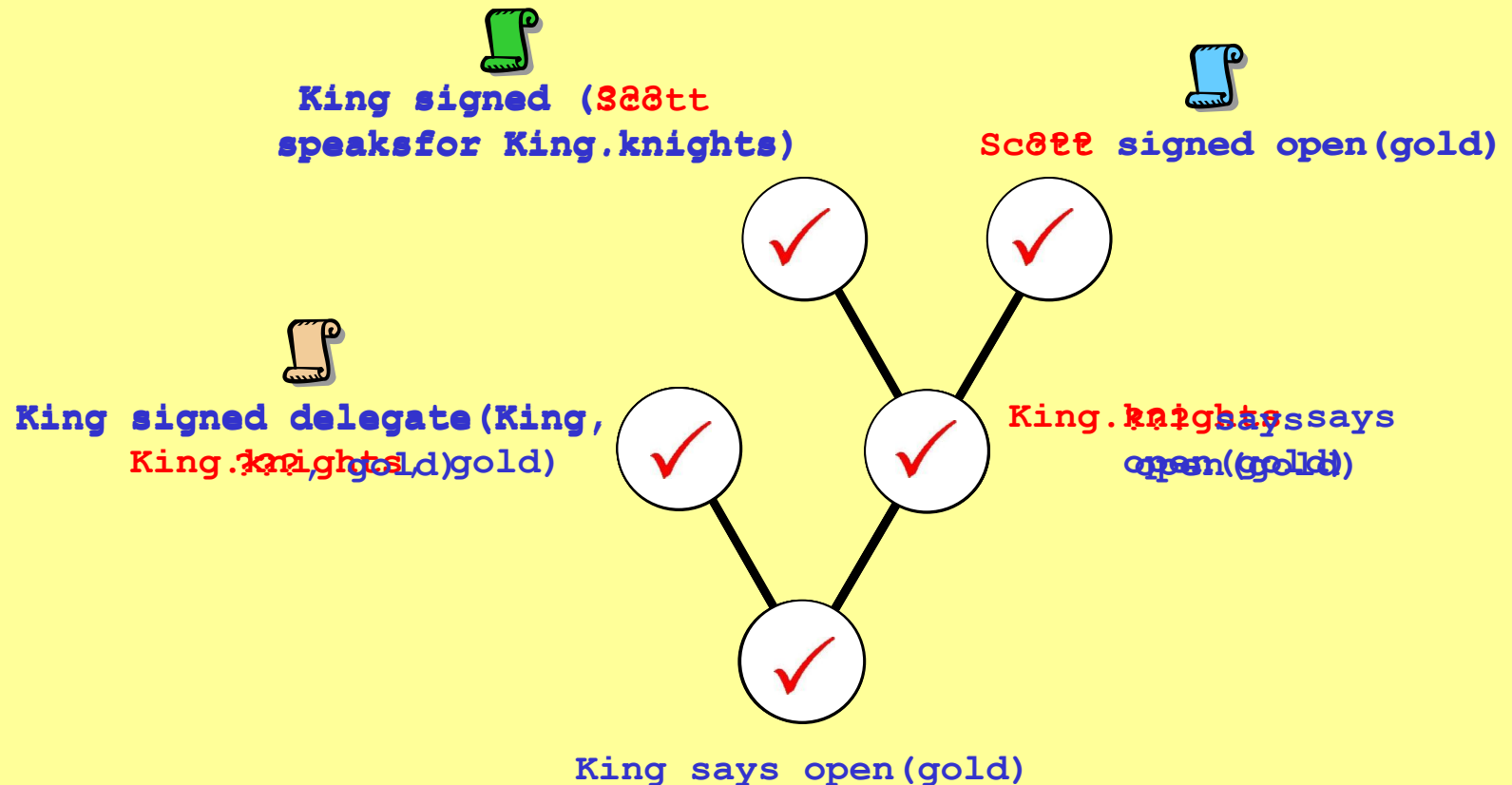
- Sample rules

$$\frac{\textit{pubkey signed } F}{\textit{key(pubkey) says } F} \quad (\textit{says-I})$$

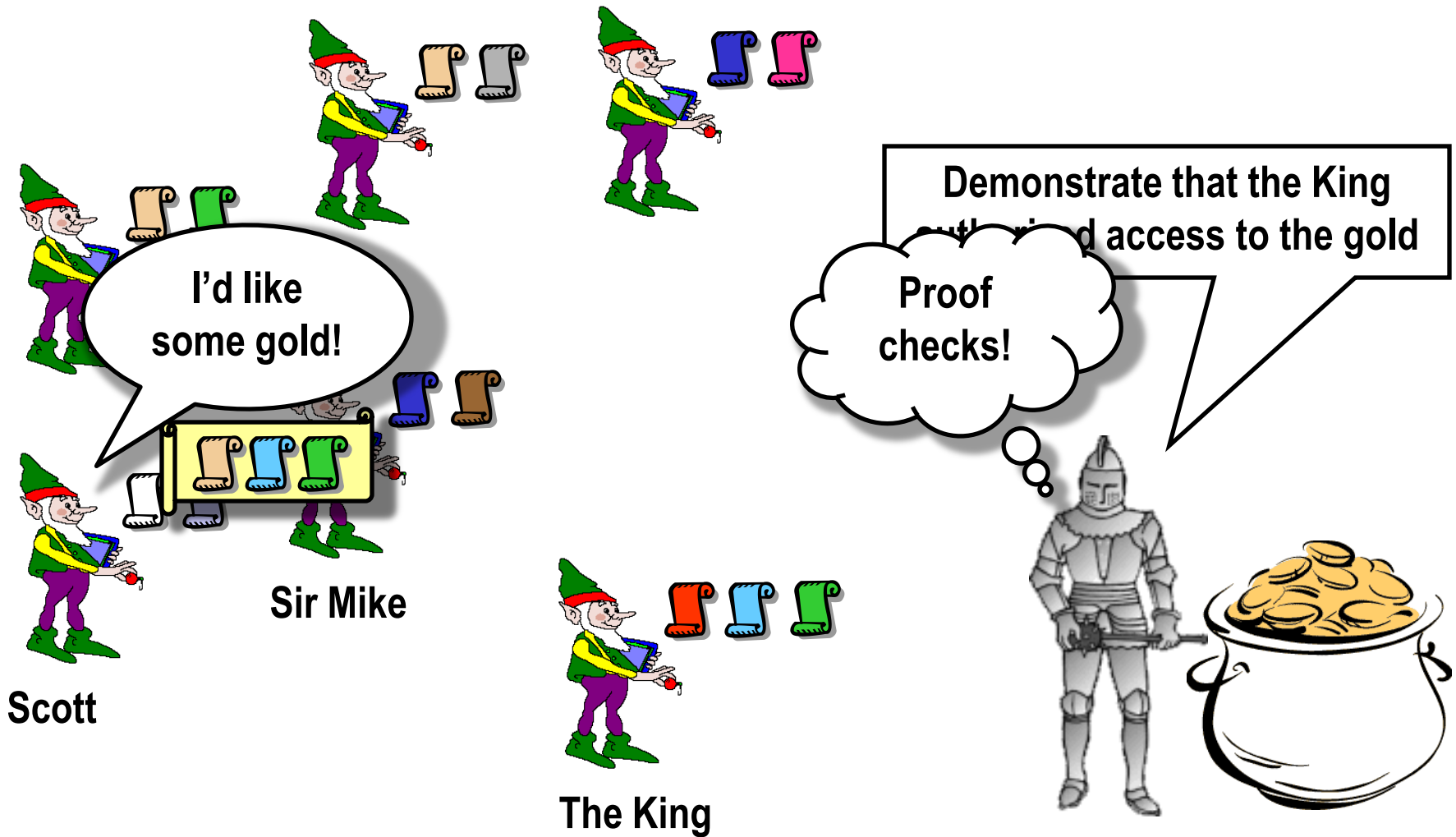
$$\frac{F}{A \textit{ says } F} \quad (\textit{says-I2})$$

$$\frac{A \textit{ says } (B \textit{ speaksfor } A) \quad B \textit{ says } F}{A \textit{ says } F} \quad (\textit{speaksfor-E})$$

# Making Access Decisions



# A Distributed Authorization Scenario



# PCA Summary

- Credentials and decision procedure are modeled/represented in formal logic
- Request to access a resource is accompanied by proof
  - Proof is easy to verify
  - Proof demonstrates compliance of access with policy
- Proof generation moved from resource monitor to client
  - Makes resource monitor's task much easier
    - In the limit, replaces an impossible task with a possible one

# Sources

- Slides from G. Necula.
- A. W. Appel and E. W. Felten. Proof-carrying authorization. In *Proceedings of the ACM Conference on Computer and Communications Security*, 1999.