

Software Foundations of Security and Privacy (15-316, spring 2017)

Lecture 7: Proof-Carrying Code

Jean Yang

`jyang2@andrew.cmu.edu`

With material from Peter Lee and David Walker (who borrowed material from Greg Morrisett)

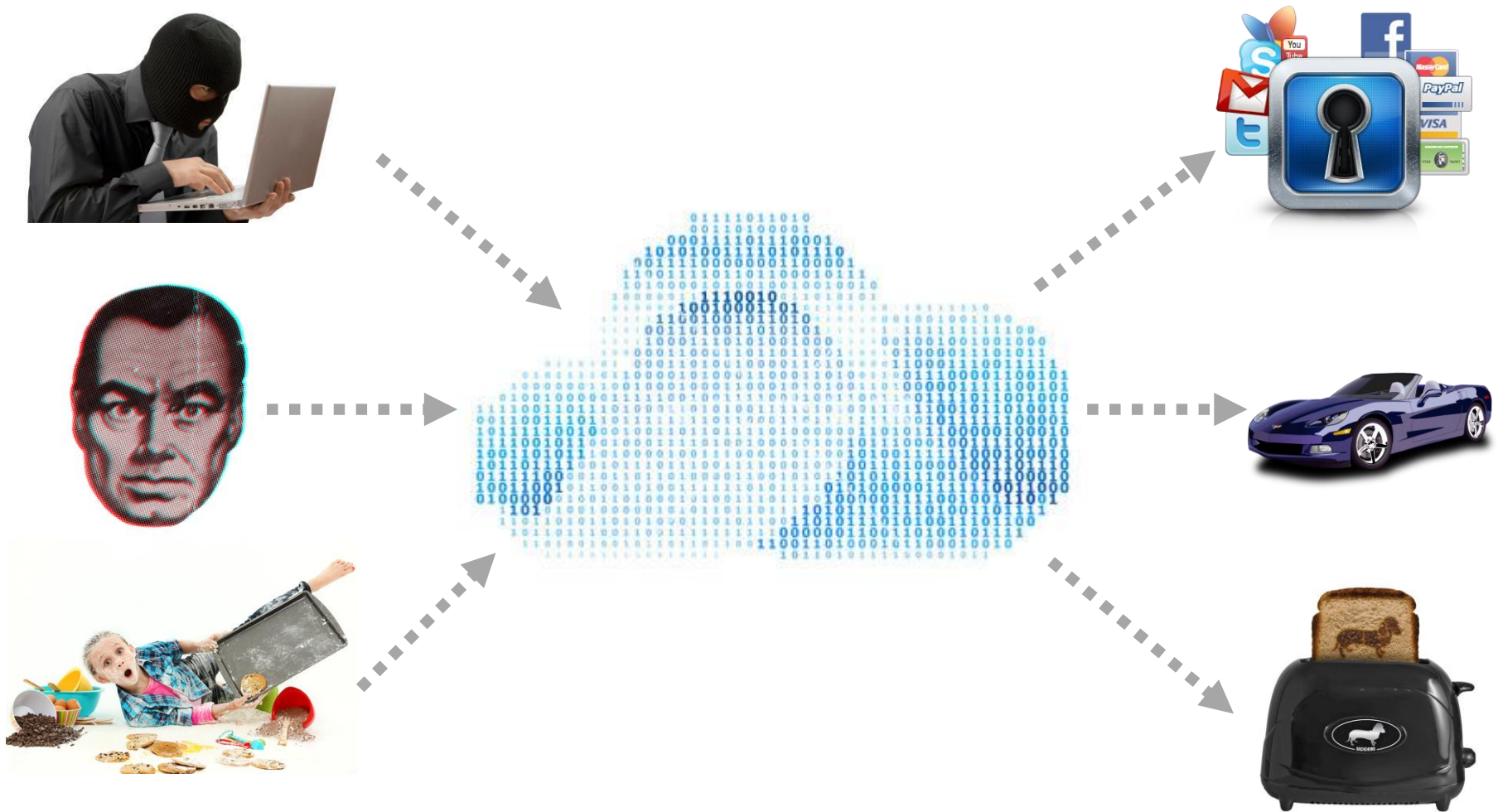
Perfectly Avoidable Dangers

“I was coming back from lunch and I spotted this USB drive on the bench over here,’ Harty said. ‘I watched for a little while and then I looked around to see if there was anybody here that was going to pick it up. I figured I needed a USB drive, so I picked it up.’” –NBC Chicago

Less Avoidable Dangers



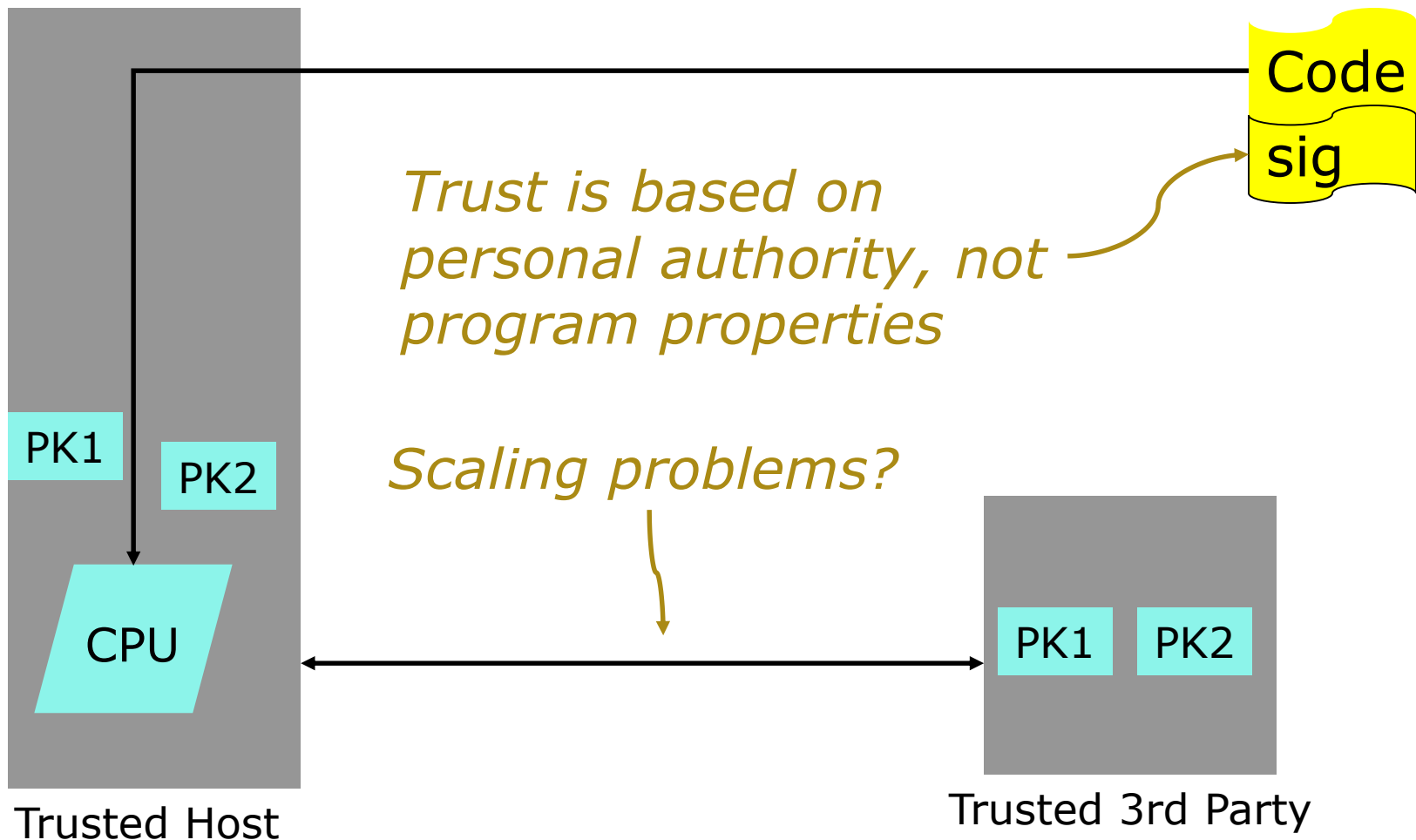
Code Safety, the #1 Existential Threat



Approach 1

Trust the code producer

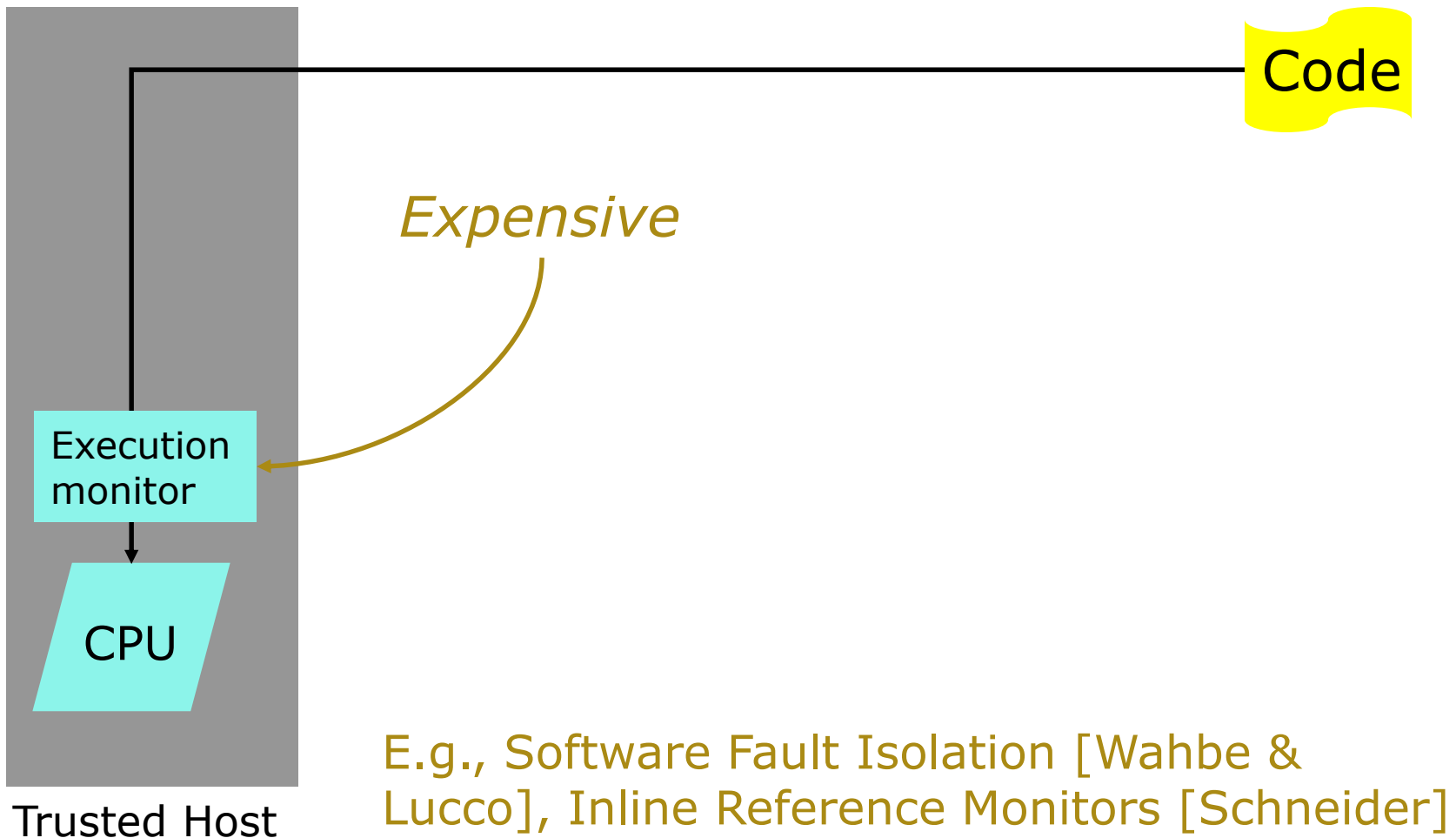
Slide from Peter Lee



Approach 2

Baby-sit the program

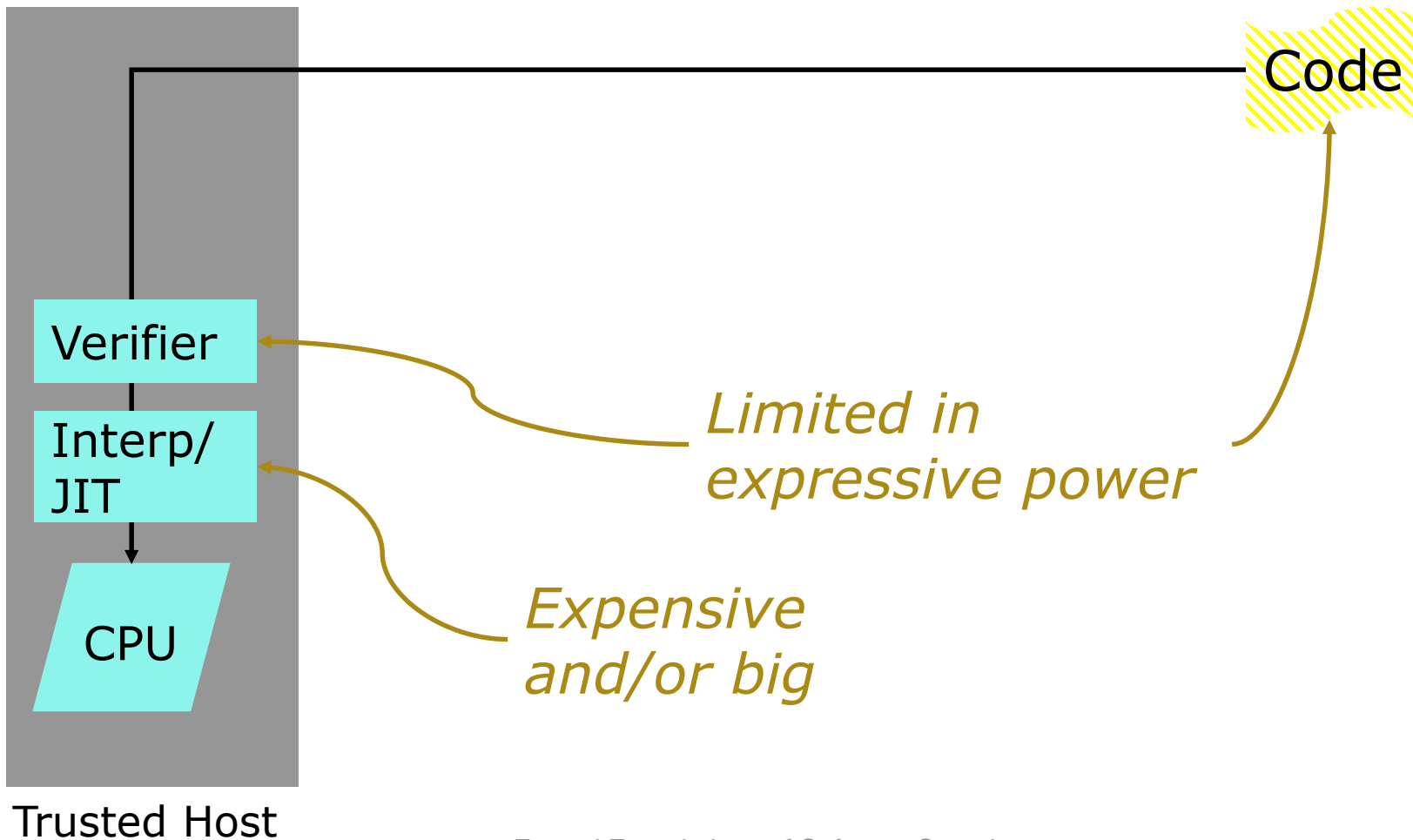
Slide from Peter Lee



Approach 3

Java

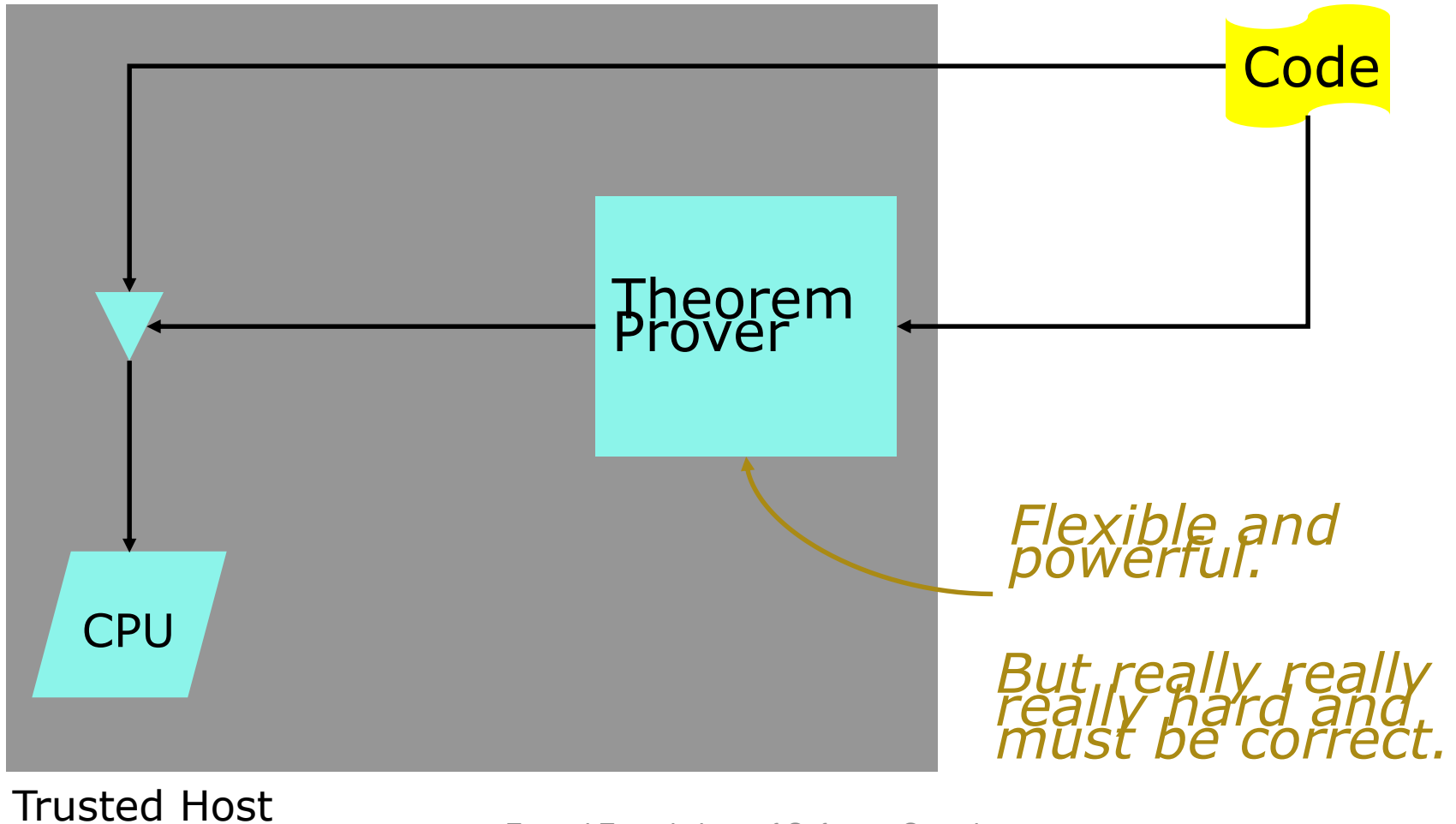
Slide from Peter Lee



Approach 4

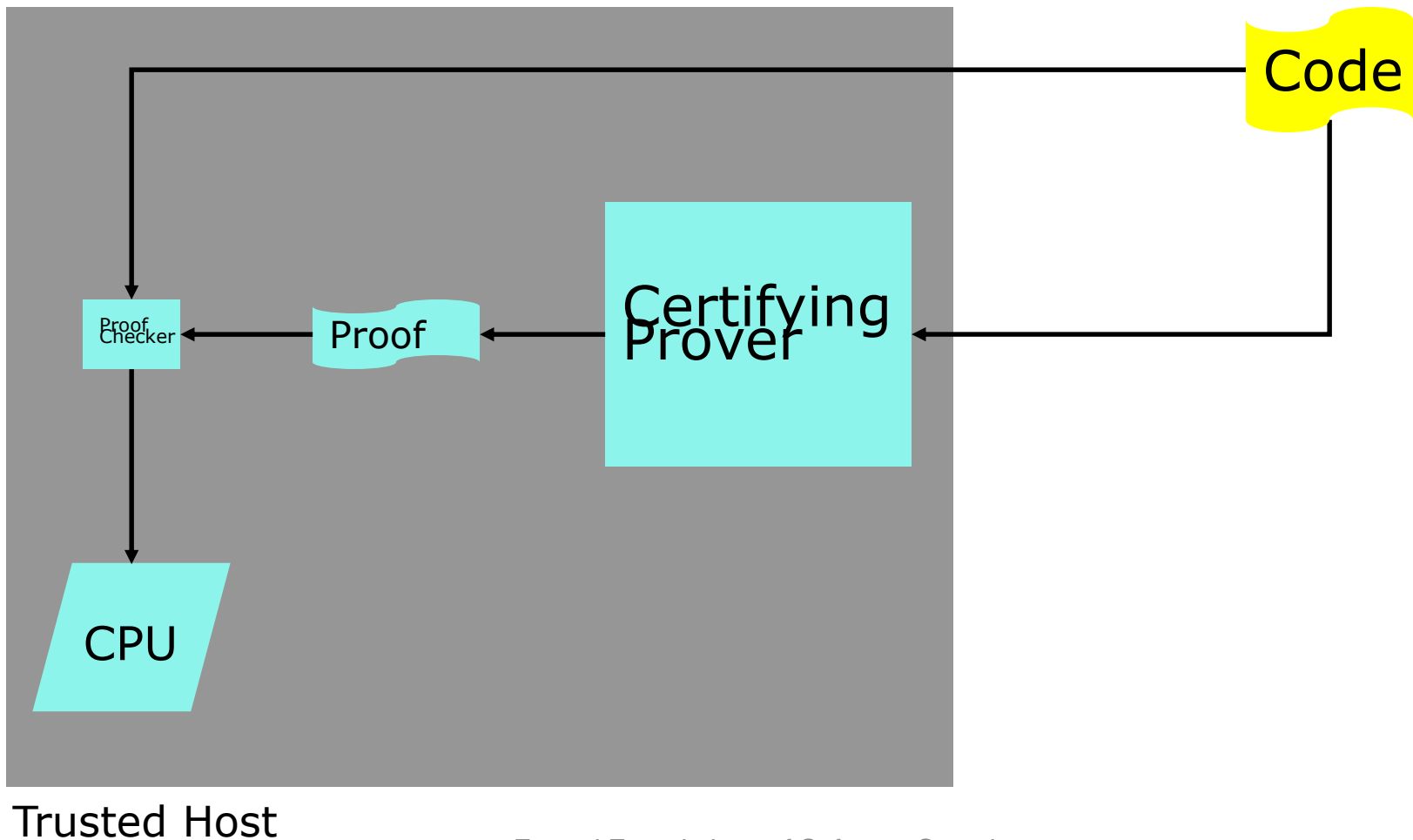
Formal verification

Slide from Peter Lee



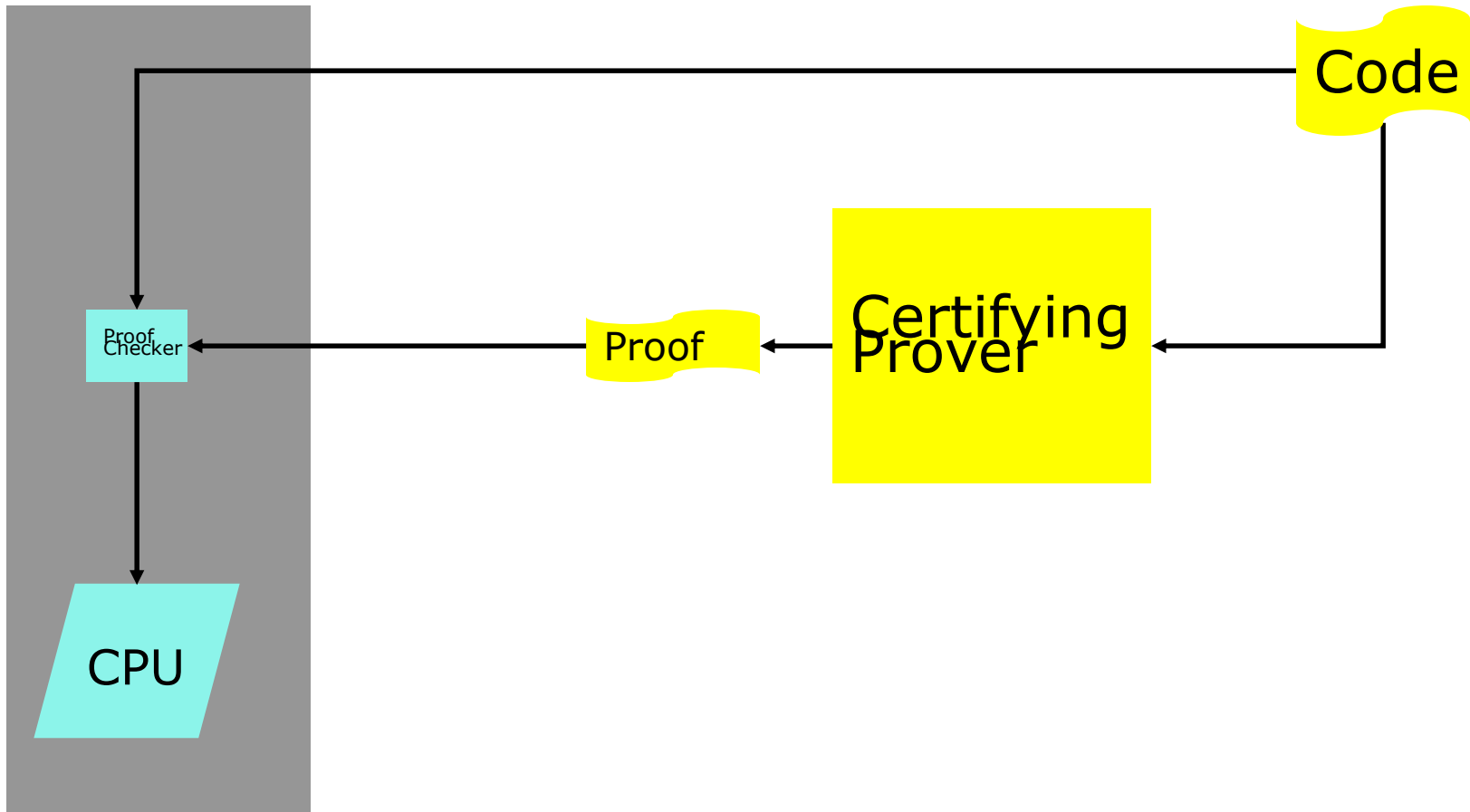
A key idea: Checkable certificates

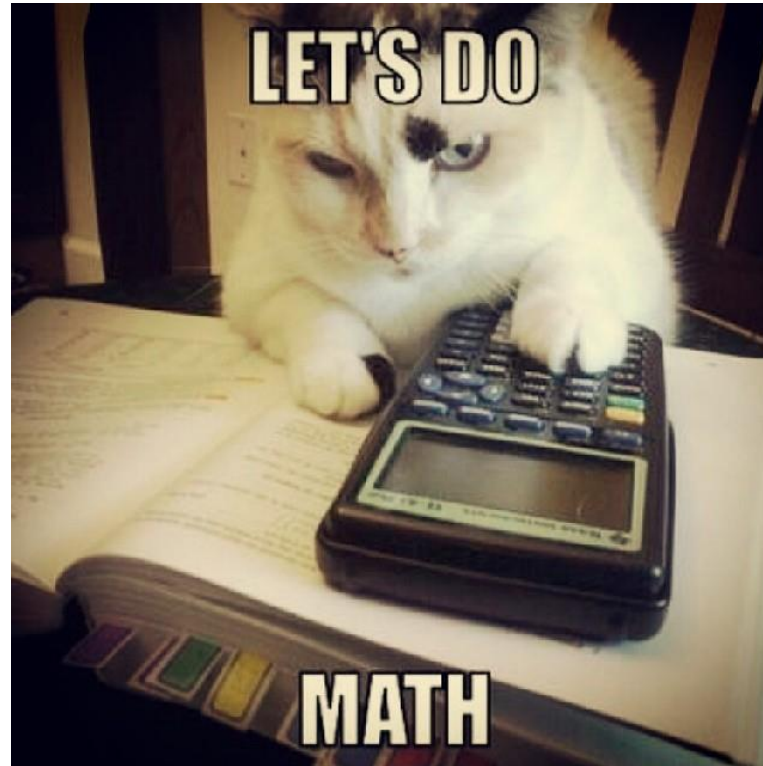
Slide from Peter Lee



A key idea: Checkable certificates

Slide from Peter Lee





Proof-Carrying Code: Five Frequently Asked Questions

[Necula and Lee, OSDI '96]

Question 1

Slide from Peter Lee

How are the proofs represented and checked?

Formal proofs

Slide from Peter Lee

Write “ x is a proof of predicate P ” as
 $x : P$.

What do proofs look like?

Example inference rule

Slide from Peter Lee

- If we have a proof x of P and a proof y of Q , then x and y together constitute a proof of $P \wedge Q$.

$$\frac{\Gamma \vdash x : P \quad \Gamma \vdash y : Q}{\Gamma \vdash (x, y) : P \wedge Q}$$

Or, in ASCII:

- Given $x:P, y:Q$ then $(x,y):P*Q$.

More inference rules

Slide from Peter Lee

- Assume we have a proof x of P . If we can then obtain a proof b of Q , then we have a proof of $P \Rightarrow Q$.
 - Given $[x:P] \ b:Q$ then
 $\text{fn } (x:P) \Rightarrow b : P \rightarrow Q$.
- More rules:
 - Given $x:P*Q$ then $\text{fst}(x) : P$
 - Given $y:P*Q$ then $\text{snd}(y) : Q$

Types and proofs

Slide from Peter Lee

So, for example:

```
fn (x:P*Q) => (snd(x) , fst(x))  
  : P*Q → Q*P
```

- *This is an ML program!*
- *Also, typechecking provides a “smart” blackboard!*

Curry-Howard Isomorphism

Slide from Peter Lee

- In a logical framework language, predicates can be represented as types and proofs as programs (i.e., expression terms).
- Furthermore, under certain conditions ***typechecking is sufficient to ensure the validity of the proofs.***

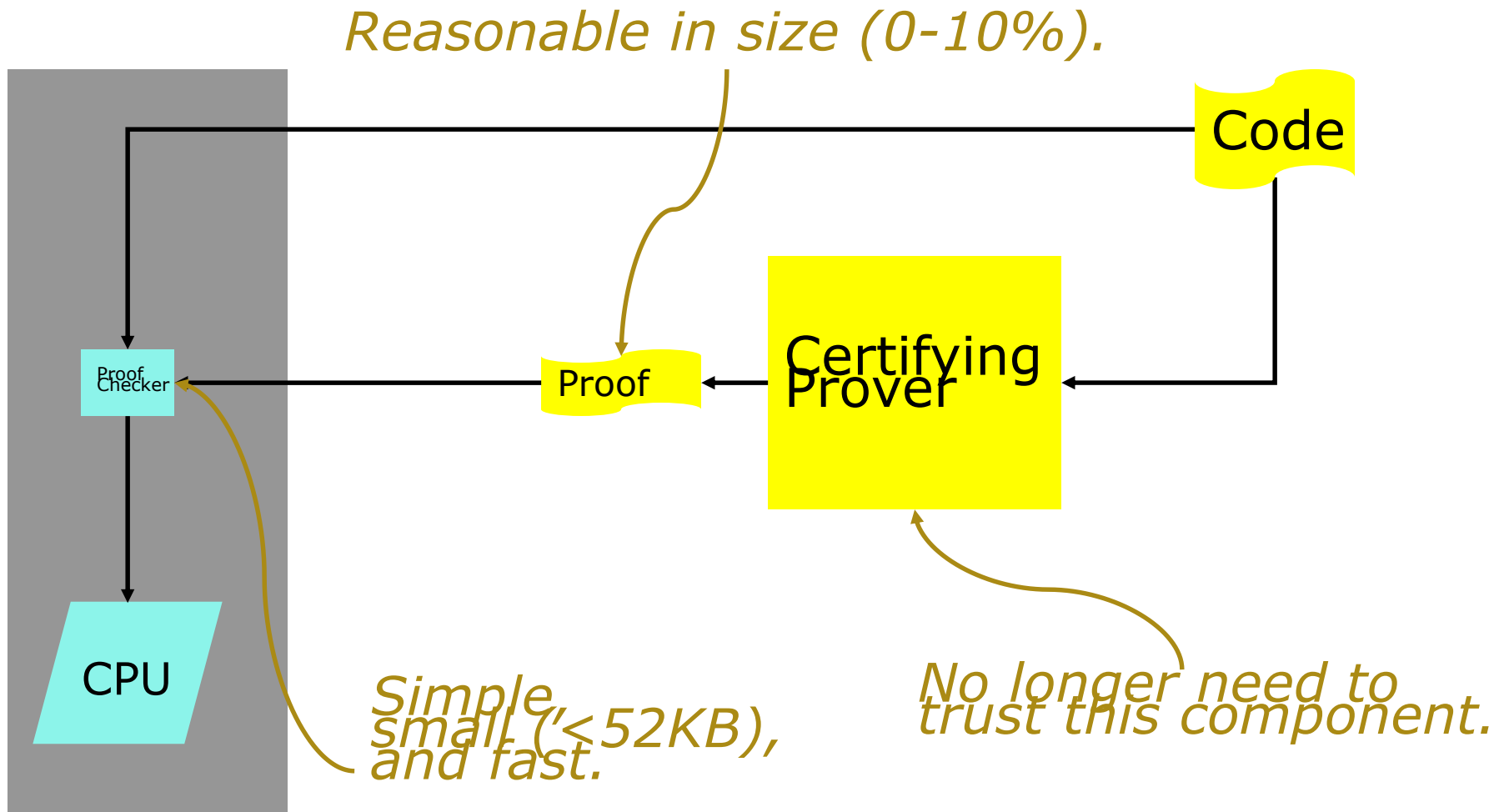
Question 2

Slide from Peter Lee

How well does this work in practice?

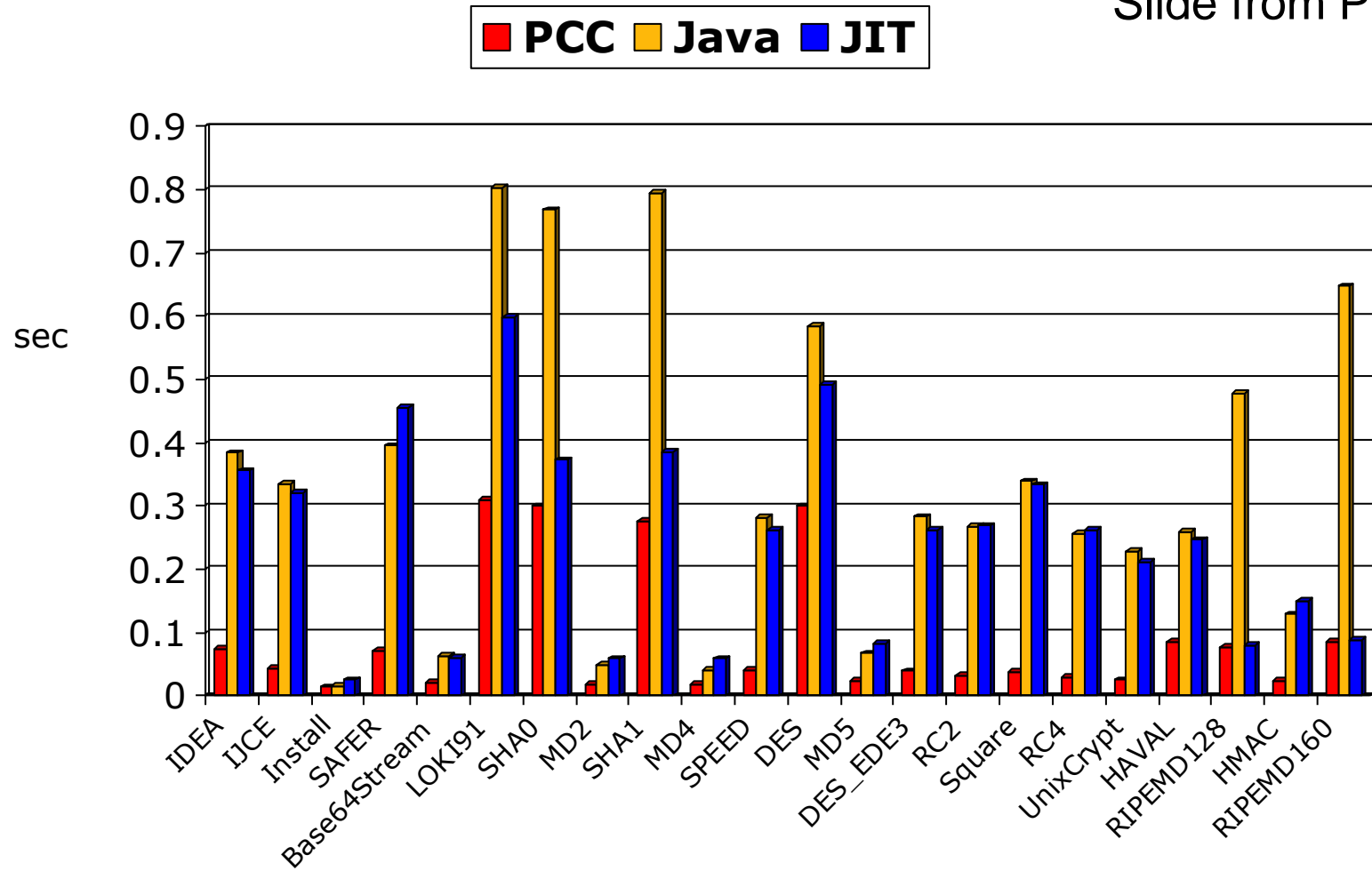
The Necula-Lee experiments

Slide from Peter Lee



Crypto test suite results

Slide from Peter Lee



Question 3

Slide from Peter Lee

- *Aren't the properties we're trying to prove undecideable?*
- *How on earth can we hope to generate the proofs?*

Solution: Programming Languages!

Slide from Peter Lee

- Civilized programming languages can provide “safety for free”
 - Well-formed/well-typed \Rightarrow safe
- **Idea**: Arrange for the compiler to “explain” why the target code it generates preserves the safety properties of the source program

Certifying Compilers

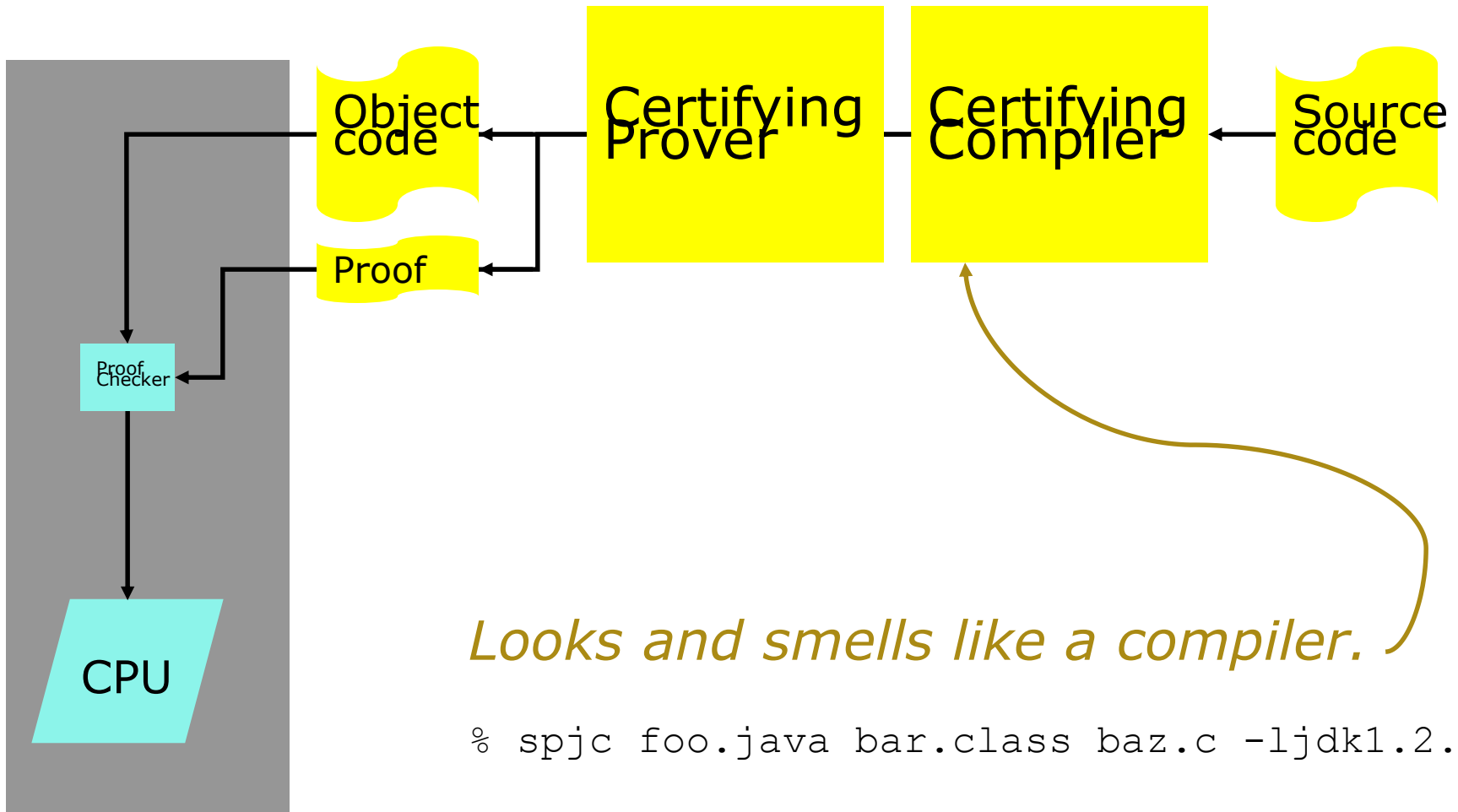
[Necula & Lee, PLDI'98]

Slide from Peter Lee

- Intuition:
 - Compiler “knows” why each translation step is semantics-preserving
 - So, have it generate a proof that safety is preserved

Certifying compilation

Slide from Peter Lee



```
% spjc foo.java bar.class baz.c -ljdk1.2.2
```


Question 4

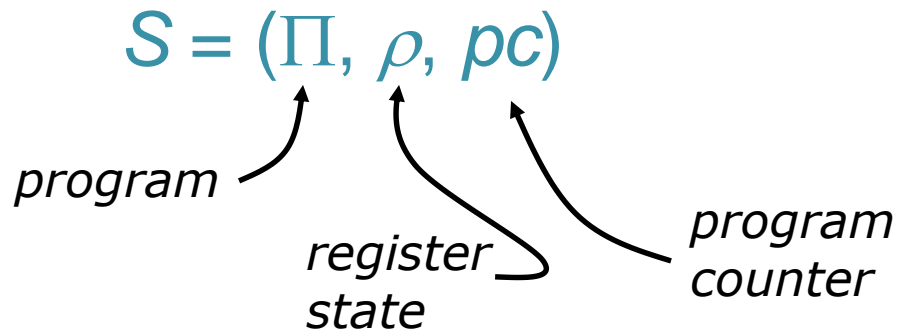
Slide from Peter Lee

- *Just what, exactly, are we proving?*
- *What are the limits?*
- *And isn't static checking inherently less powerful than dynamic checking?*

Semantics

Slide from Peter Lee

Define the states of the target machine



and a transition function $\text{Step}(S)$.

Define also the safe machine states via the *safety policy* $\text{SP}(S)$.

Semantics, cont'd

Slide from Peter Lee

Then we have the following predicate for safe execution:

$$\text{Safe}(S) = \prod n:\text{Nat}. \text{SP}(\text{Step}^n(S))$$

and proof-carrying code:

$$\text{PCC} = (S_0:\text{State}, P:\text{Safe}(S_0))$$

Reference Interpreters

Slide from Peter Lee

- A *reference interpreter (RI)* is a standard interpreter extended with instrumentation to check the safety of each instruction before it is executed, and abort execution if anything unsafe is about to happen.
- In other words, an RI is capable *only* of safe execution.

Reference Interpreters

cont'd

Slide from Peter Lee

- The reference interpreter is never actually implemented.
- The point will be *to prove that execution of the code on the RI never aborts, and thus execution on the real hardware will be identical to execution on the RI.*

Questions

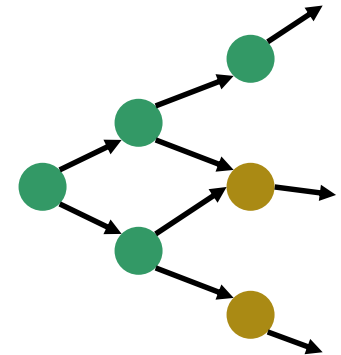
Slide from Peter Lee

- Suppose that we require the code to execute no more than N instructions. Is such a safety property enforceable by an RI?
- Suppose we require the code to terminate eventually. Is such a safety property enforceable by an RI?

What can't be enforced?

Slide from Peter Lee

- Safety properties \Rightarrow *Yes*
 - “No bad thing will happen”
- Liveness properties \Rightarrow *Not yet*
 - “A good thing will eventually happen”



Static vs dynamic checking

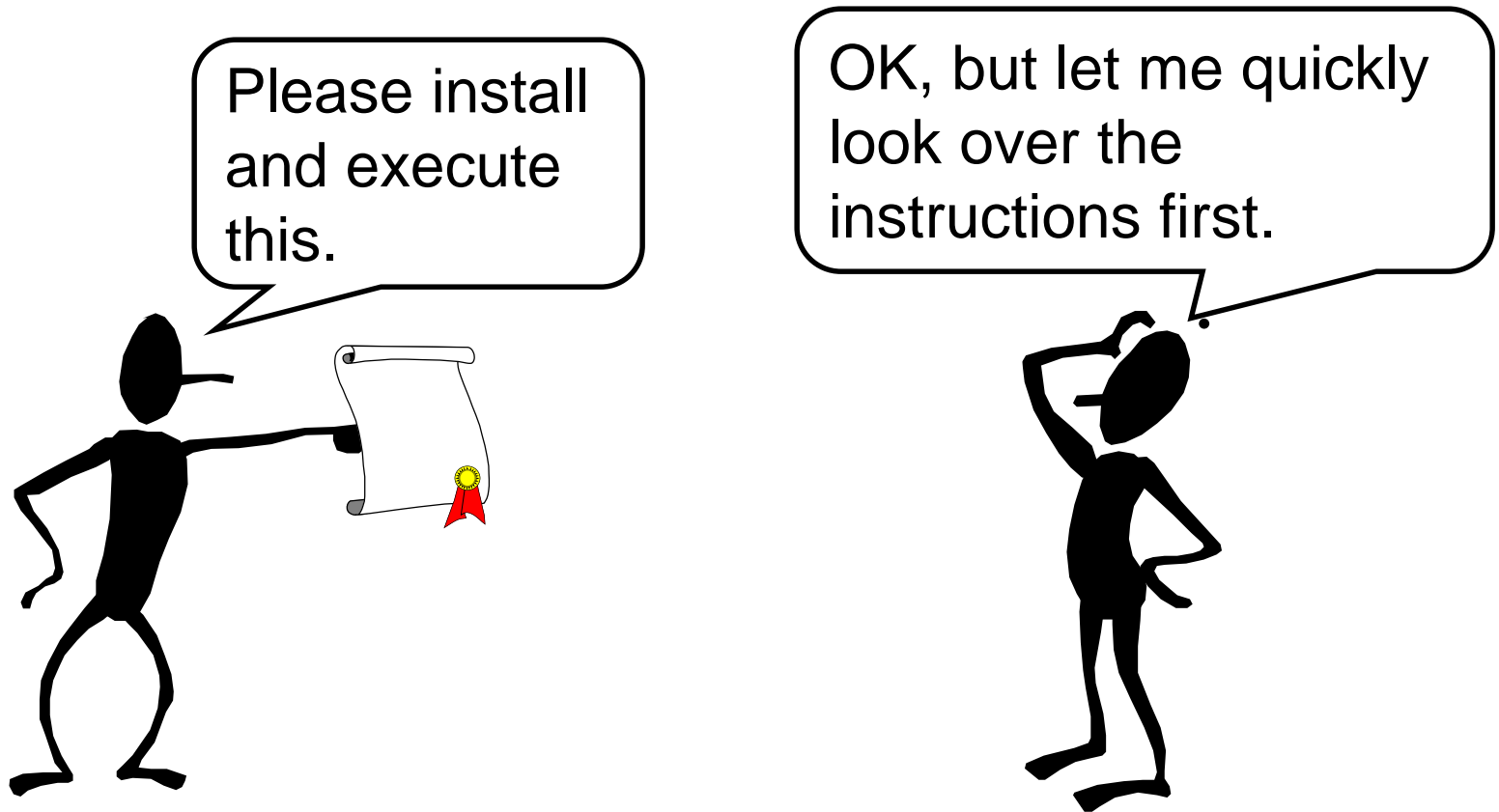
Slide from Peter Lee

- PCC provides a basis for static enforcement of safety conditions
- However, PCC is not just for static checking
- PCC can be used, for example, to verify that necessary dynamic checks are carried out properly

Question 5

Slide from Peter Lee

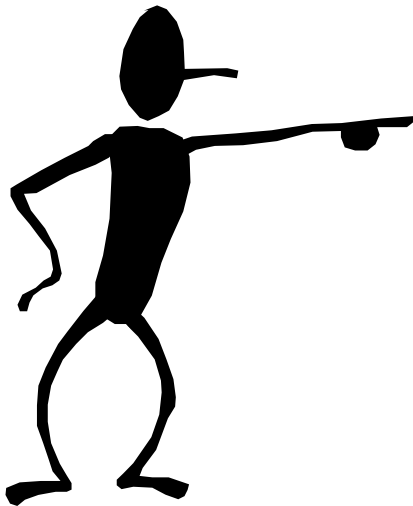
Even if the proof is valid, how do we know that it is a safety proof of the given program?



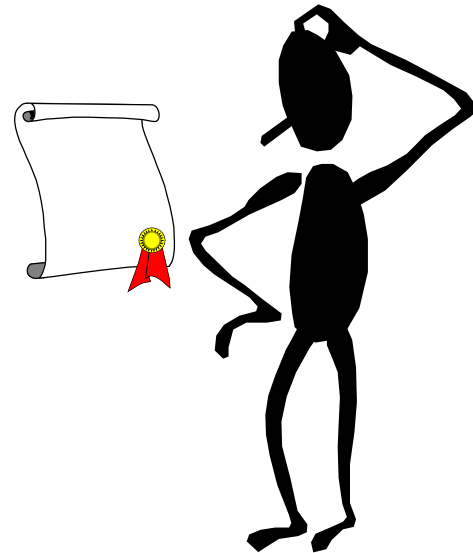
Code producer

Formal Foundations of Software Security

Host



Code producer

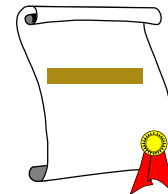


Host

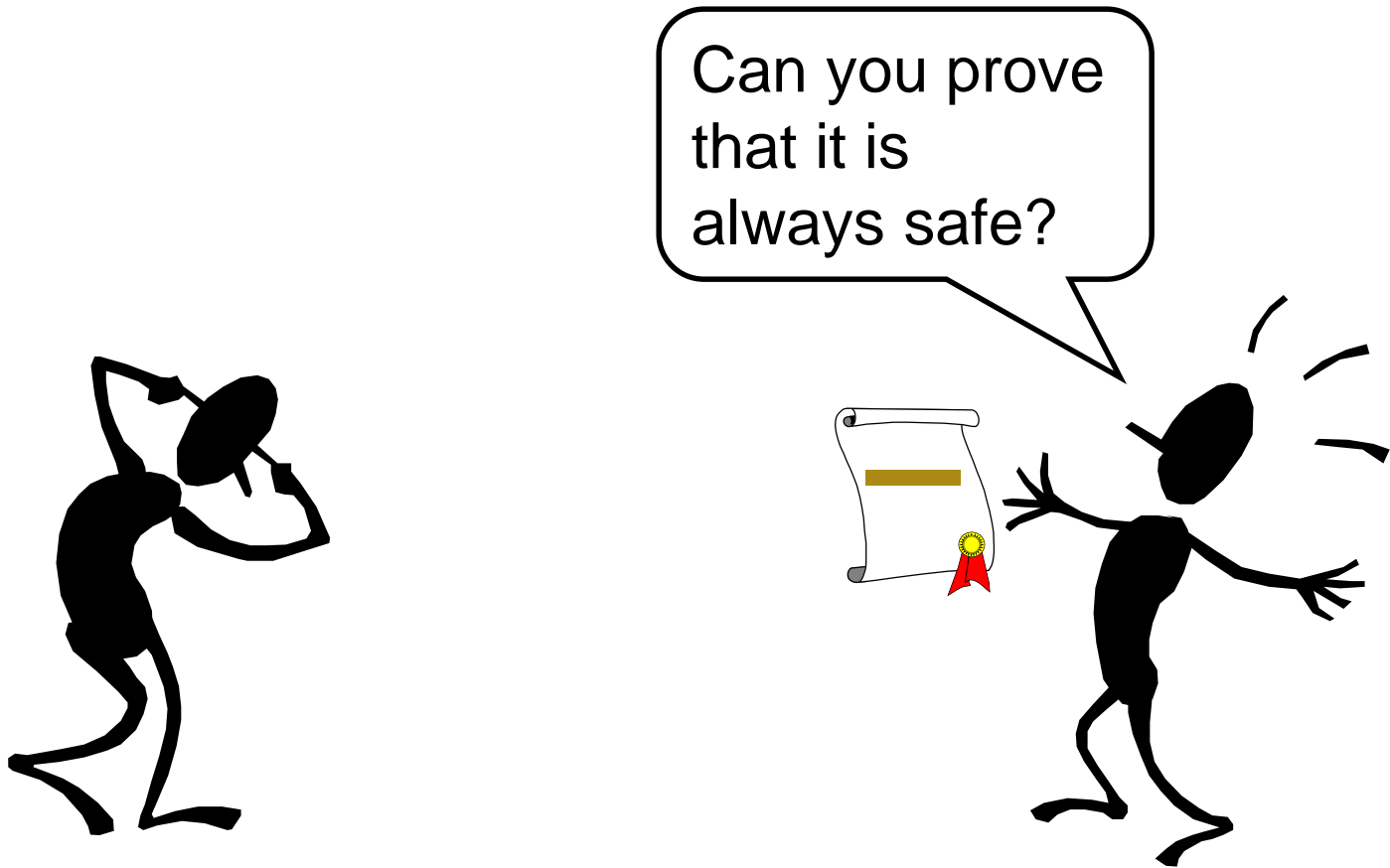


Code producer

This `store`
instruction is
dangerous!



Host

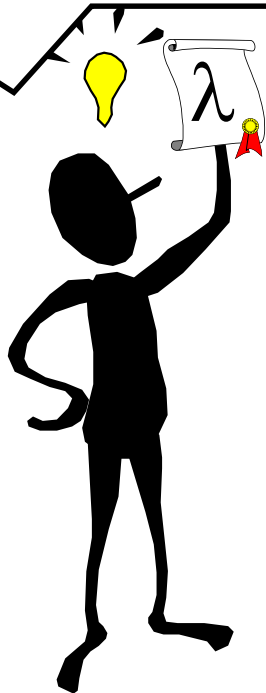


Code producer

Formal Foundations of Software Security

Host

Yes! Here's the proof I got from my certifying Java compiler!

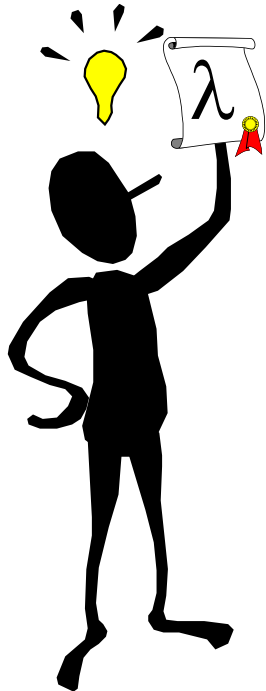


Can you prove that it is always safe?



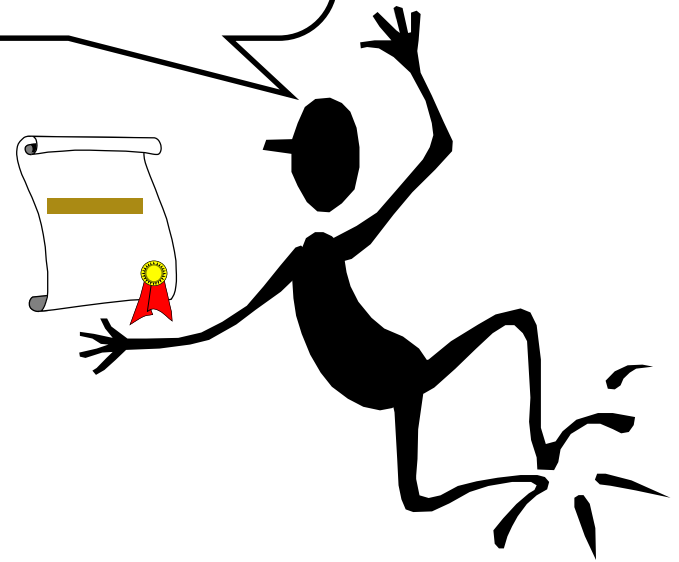
Code producer

Host



Code producer

Your proof checks out. I believe you because I believe in logic.



Host

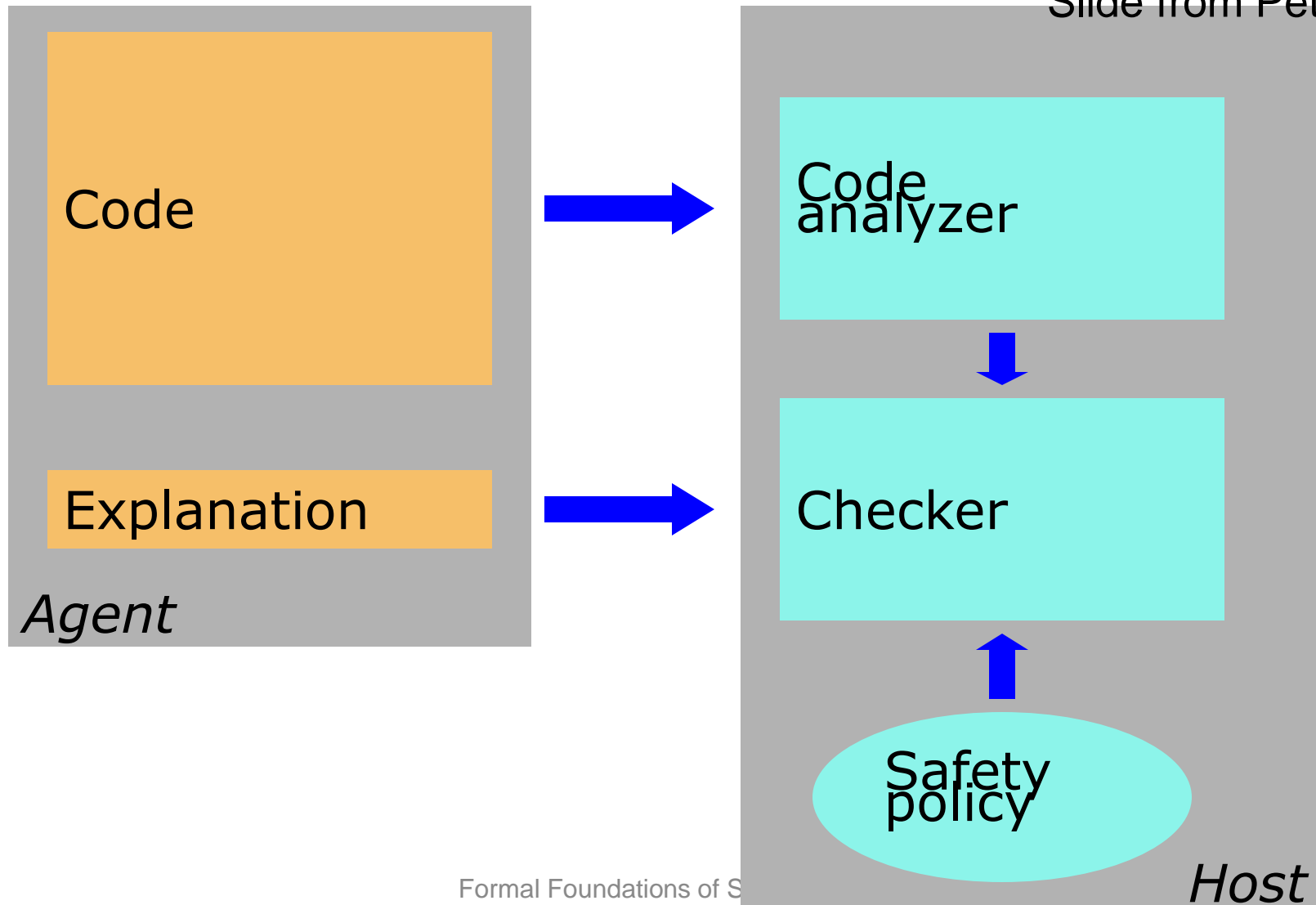
The safety policy

Slide from Peter Lee

- We need a method for
 - identifying the dangerous instructions, and
 - generating logical predicates whose validity implies that the instruction is safe to execute
- In practice, we will also need
 - specifications (pre/post-conditions) for each required entry point in the code, as well as the trusted API.

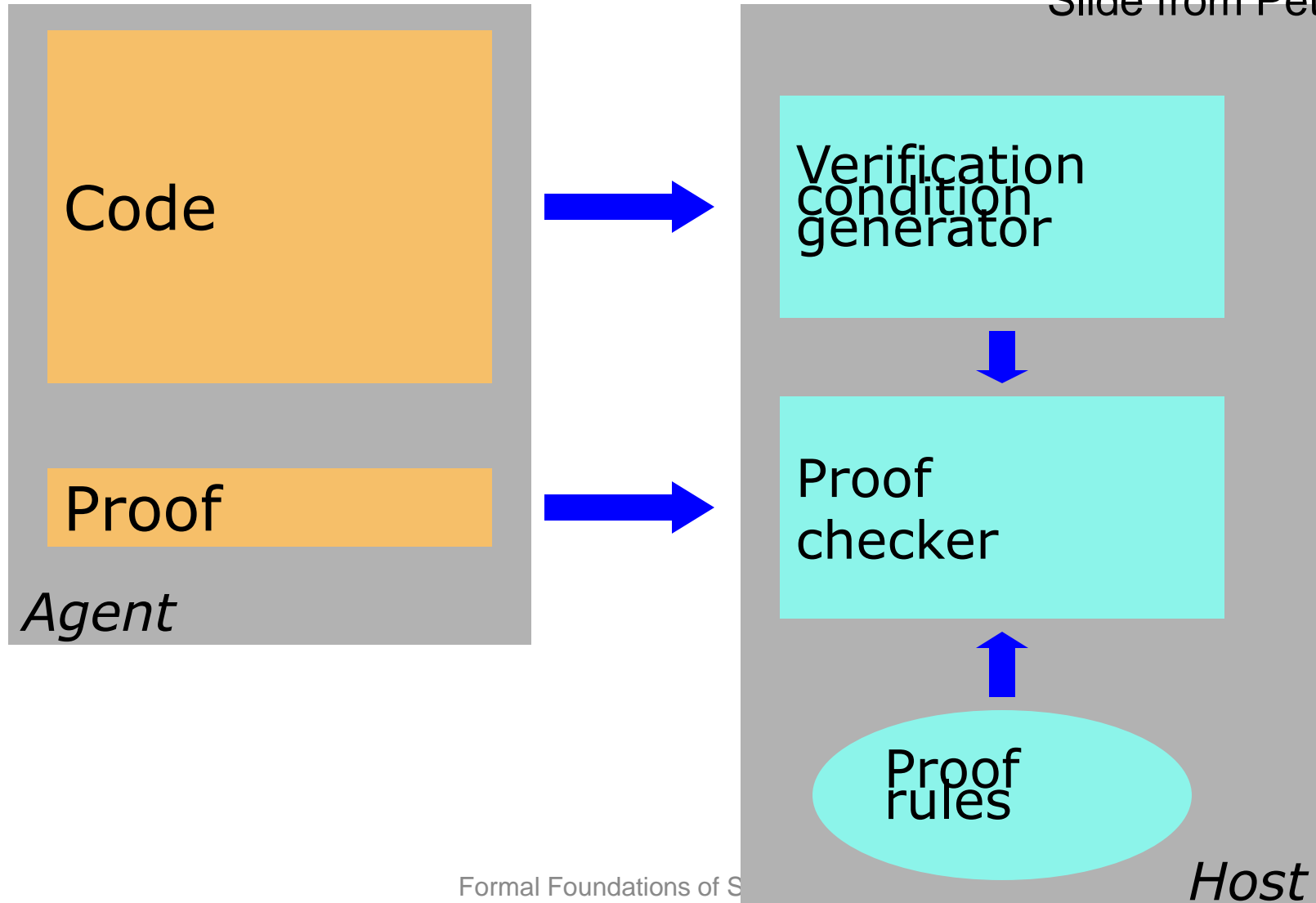
High-level architecture

Slide from Peter Lee



High-level architecture

Slide from Peter Lee





RAWR I AM A SINK

Case Study: Program Resource Usage

A case study


Slide from Peter Lee

As a case study, let us consider the problem of verifying that programs do not use more than a specified amount of some resource.

```
s ::= skip
    | i := e
    | if e then s else s
    | while e do s
    | s ; s
    | use e
```

```
e ::= n
    | i | read()
    | e + e | e - e | ...
```

Denotes the use of n pieces of the resource, where e evaluates to n



Case study, cont'd

Slide from Peter Lee

Under normal circumstances, one would implement the statement:

use e ;

in such a way that every time it is executed, a run-time check is performed in order to determine whether n pieces of the resource are available (assuming e evaluates to n).

Case study, cont'd

Slide from Peter Lee

However, this stinks because many times we should be able to infer that there are definitely available resources.

If somehow we know that there are ≥ 9 available here...

```
...  
if ...  
    then use 4;  
    else use 5;  
use 4;  
...
```

...then certainly there is no need to check any of these uses!

An easy (well, probably) case

Slide from Peter Lee

```
Program Static
  i := 0
  while i < 10000
    use 1
    i := i + 1
```

*We ought to be able to
prove statically whether
the uses are safe*

A hopeless case

Slide from Peter Lee

```
Program Dynamic  
  while read() != 0  
    use 1
```


An interesting case

Slide from Peter Lee

```
Program Interesting
  N := read()
  i := 0
  while i < N
    use 1
    i := i + 1
```

*In principle, with just a single
dynamic check, static proof
ought to be possible*

Also interesting

Slide from Peter Lee

```
Program AlsoInteresting
  while read() != 0
    i := 0
    while i < 100
      use 1
      i := i + 1
```

A core principle of PCC

Slide from Peter Lee

In the code, the implementation of a safety-critical operation should be *separated* from the implementation of its safety checks.

Separating use from check

Slide from Peter Lee

- So, what we would like to do is to separate the safety check from the use.
- We do this by introducing a new construct, **acquire**
- **acquire** requests n amount of resource; **use** no longer does any checking

```
s ::= skip
    | i := e
    | if e then s else s
    | while e do s
    | s ; s
    | use e
    | acquire e
```

DON'T STOP...



PCC to the Limit: Typed Assembly

Typed Assembly

- Assembly code is annotated with type information that can be verified by our type checker.
- Type soundness implies many important security properties such as memory safety.
- TAL applets, like Java applets, may be downloaded from untrusted sources on the internet, verified, and executed safely without fear they will corrupt the host machine.
- Because TAL is assembly language, there is no interpretation overhead during this process and no just-in-time compiler to run or trust.

Typed Assembly

Slide from David Walker

A type system for assembly language(s):

- built-in abstractions (tuple, code)
- operators to build new abstractions ($\forall, \exists, \lambda$)
- annotations on assembly code
- an abstraction checker

Thm: well-annotated code cannot violate abstractions.

TAL Work_[popl 98, toplas 99 & others]

Slide from David Walker

Theory:

- small RISC-style assembly language
- compiler from System F to TAL
- soundness and preservation theorems

Practice:

- most of IA32 (32-bit Intel x86)
- more type constructors
 - everything you can think of and more
- safe C compiler
 - ~40,000LOC & compiles itself

Why Type Assembly?

Slide from David Walker

Theory:

- simplifies proofs of compiler correctness
- deeper understanding of compilation

Practice:

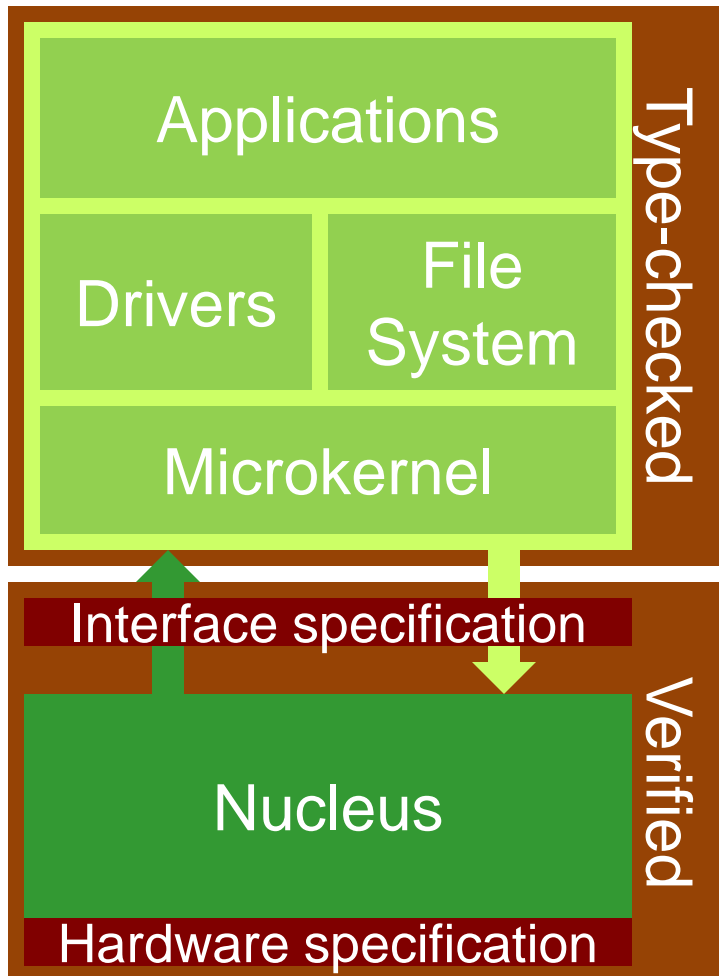
- compiler debugging
- software-based protection

Simple Built-In Types

Slide from David Walker

- Bytes: $b1, b2, b4$
- Tuples: $(\tau_1^{\phi_1}, \dots, \tau_n^{\phi_n})$ ($\phi = 0, 1$)
- Code: $\{r_1:\tau_1, \dots, r_n:\tau_n\}$
 - like a pre-condition
 - argument type of function
 - no return type because code doesn't really return, just jumps somewhere else...
- Polymorphic types: $\forall \alpha. \tau, \exists \alpha. \tau$

Circling Back: Verve



- High-level C# code compiles to **typed assembly**.
- Verve is verified against an interface to TAL for **end-to-end safety**.

Discussion Questions

- What kinds of properties can proof-carrying code help ensure?
- How does proof-carrying code relate to reference monitors?
- What guarantees do we get if we have a piece of proof-carrying code?
- What are the restrictions that proof-carrying code places on code?

Out Today: Assignment 2

- **Main idea:** implement a simple reference monitor with read/write/append/delegate permissions.
- You will need to specify the *security automaton*, as well as the *mediation points*.
- There will also be an exercise on *bounded availability* testing your understanding of safety!