Language Techniques for Provably Safe Mobile Code

Frank Pfenning
Carnegie Mellon University

Distinguished Lecture Series
Computing and Information Sciences
Kansas State University

October 27, 2000

Acknowledgments: Karl Crary, Robert Harper, Peter Lee, Greg Morrisett, George Necula, . . .

Outline

1. The Safety Problem
2. The Trusted Computing Base
3. Typed Assemby Language (TAL)
4. Proof-Carrying Code (PCC)
5. Conclusion

Mobile Code

- Java applets
- Browser plugins
- Device drivers and packet filters
- MacOS extensions
- Spreadsheet macros
- PostScript files
- ... *your favorite example* ...

Program Properties

- Complex, tightly interacting software systems.
- How do we achieve *safety* (no crash-and-burn)?
- How do we achieve *security* (no unauthorized access)?
- How do we achieve *correctness* (satisfies specification)?
- This talk concentrates on safety.

Safety Problem Solved!

- Milner's slogan: *well-typed programs cannot go wrong*.
- This is a **theorem** about the ML programming language!
- Corresponding theorems for Java, Scheme, and others.
  safe languages
- Achieved with compile time and run time checking.
- False for C and C++ (e.g. no bounds checking on arrays).
  unsafe languages

Safety Problem Solved?

- Distance between mathematical model of high-level programming language and machine execution.
- Central question:

  How do we bridge this gap to allow
  program composition and safe, efficient execution?

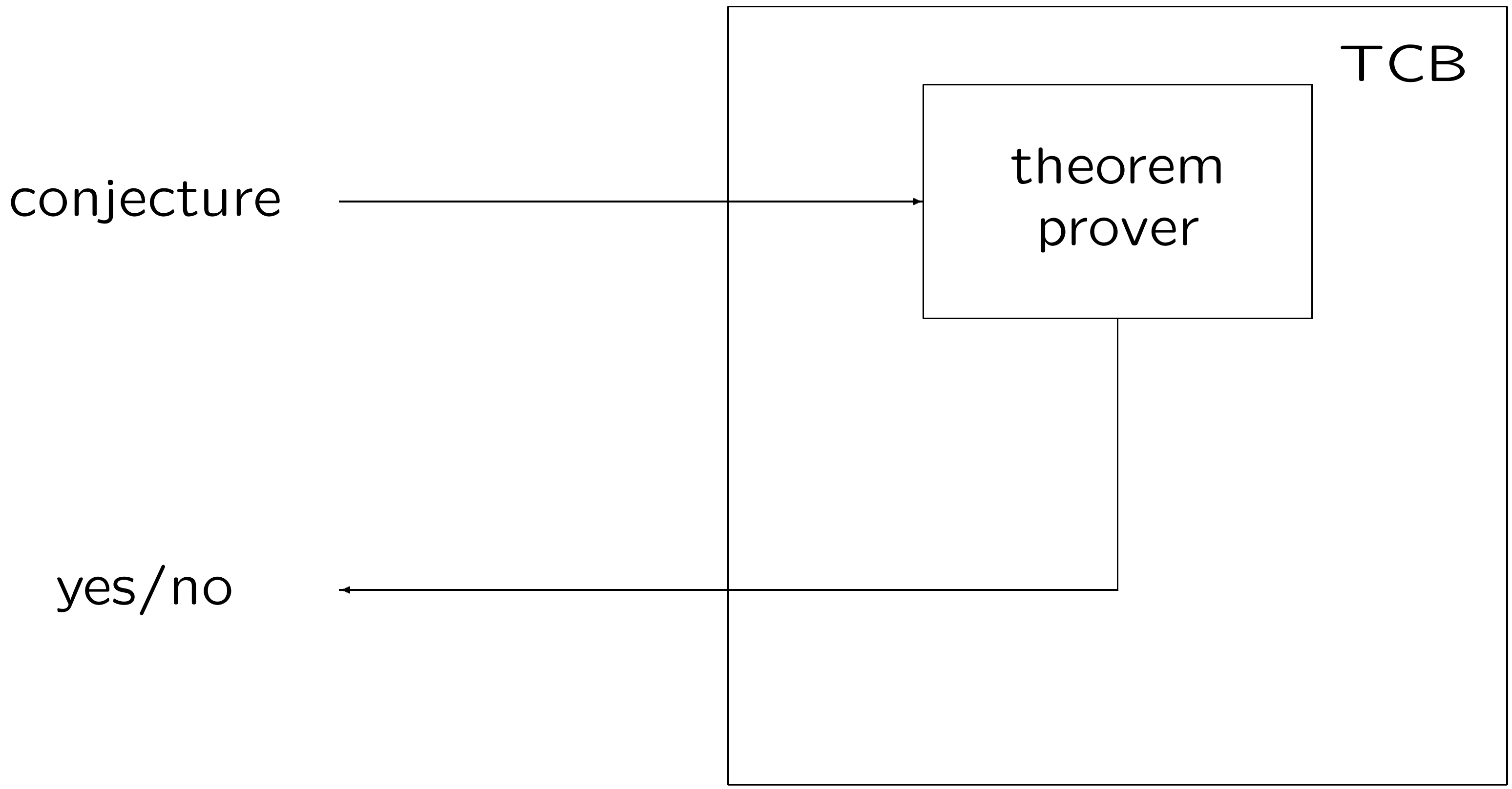
- We will not discuss:
  - Authentication and security.
  - Digital signatures and assigning blame.

The Trusted Computing Base

- Examine various safety architectures.
- Overhead in code size?
- Overhead in efficiency?
- Complexity of the *trusted computing base* (TCB)?

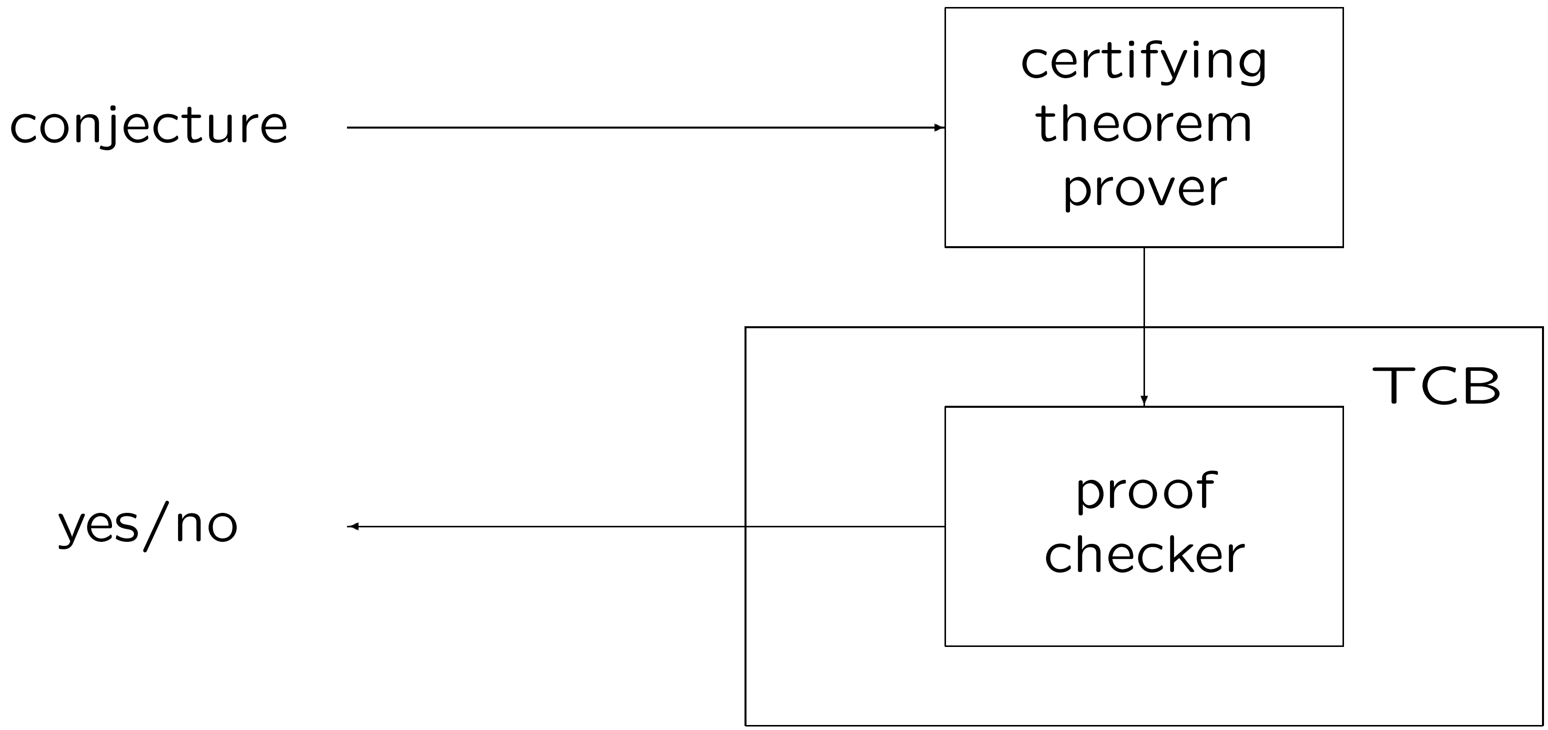
Which components do we have to trust in order to believe in the safety of the whole system?

TCB Example — Theorem Proving



- Theorem provers are complex.

TCB Example — Proof Checking



- Proof checker is much simpler than theorem prover.
- Proof checker is much easier to trust.

Applications — Proof Checking

- Resolution to type theory (Coq). [de Nivelle'99]
- Model checker (SVC2) to logical framework (Twelf). [Stump & Dill'99]
- Nelson-Oppen cooperating decision procedures to logical framework (Twelf). [Necula'98]
- Important software engineering tool!
- Logical framework (LF) as generic proof-checking engine. [Harper, Honsell & Plotkin'87] [Pf.'91]

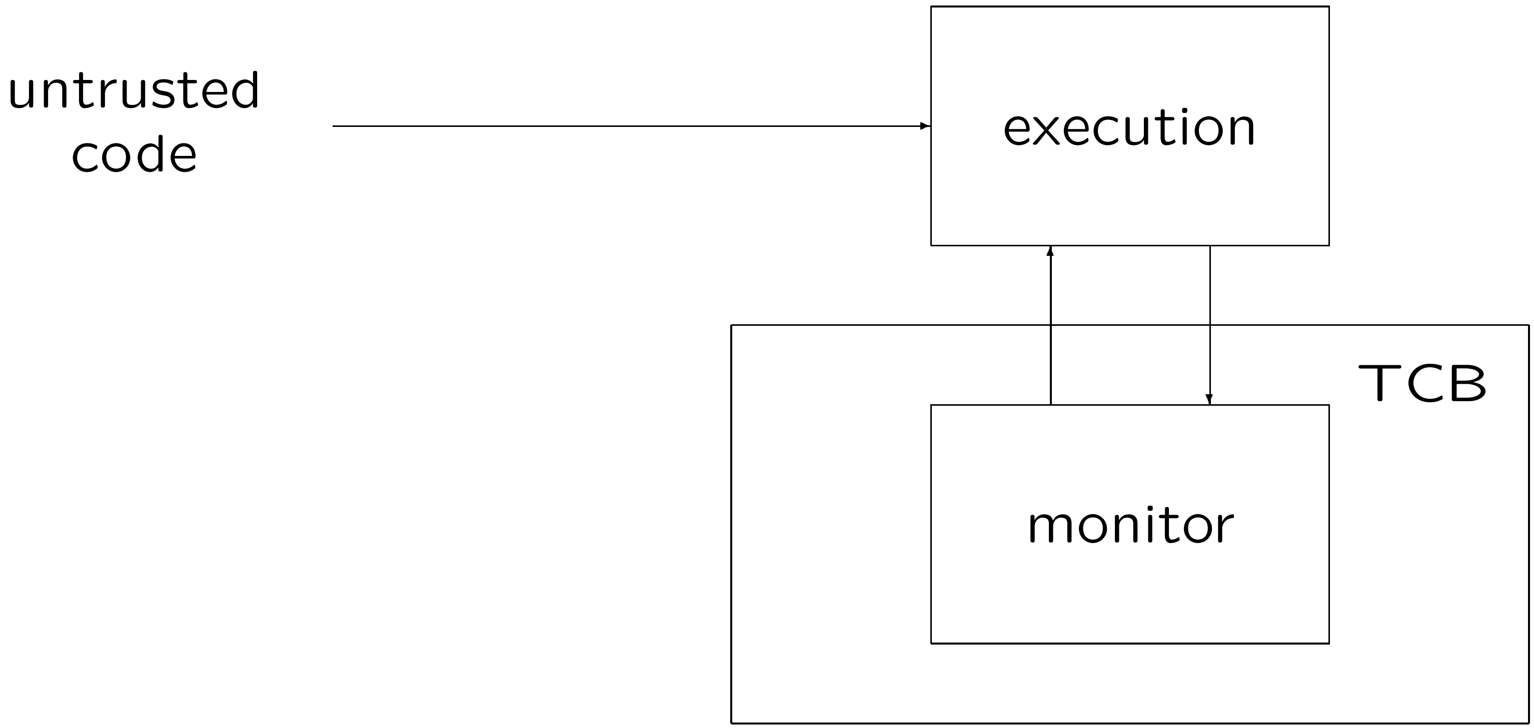
Back to Mobile Code

- Safety policies
- Reference monitor
- Software Fault Isolation (SFI)
- Typed Assembly Language (TAL)
- Proof-Carrying Code (PCC)

Safety Policies

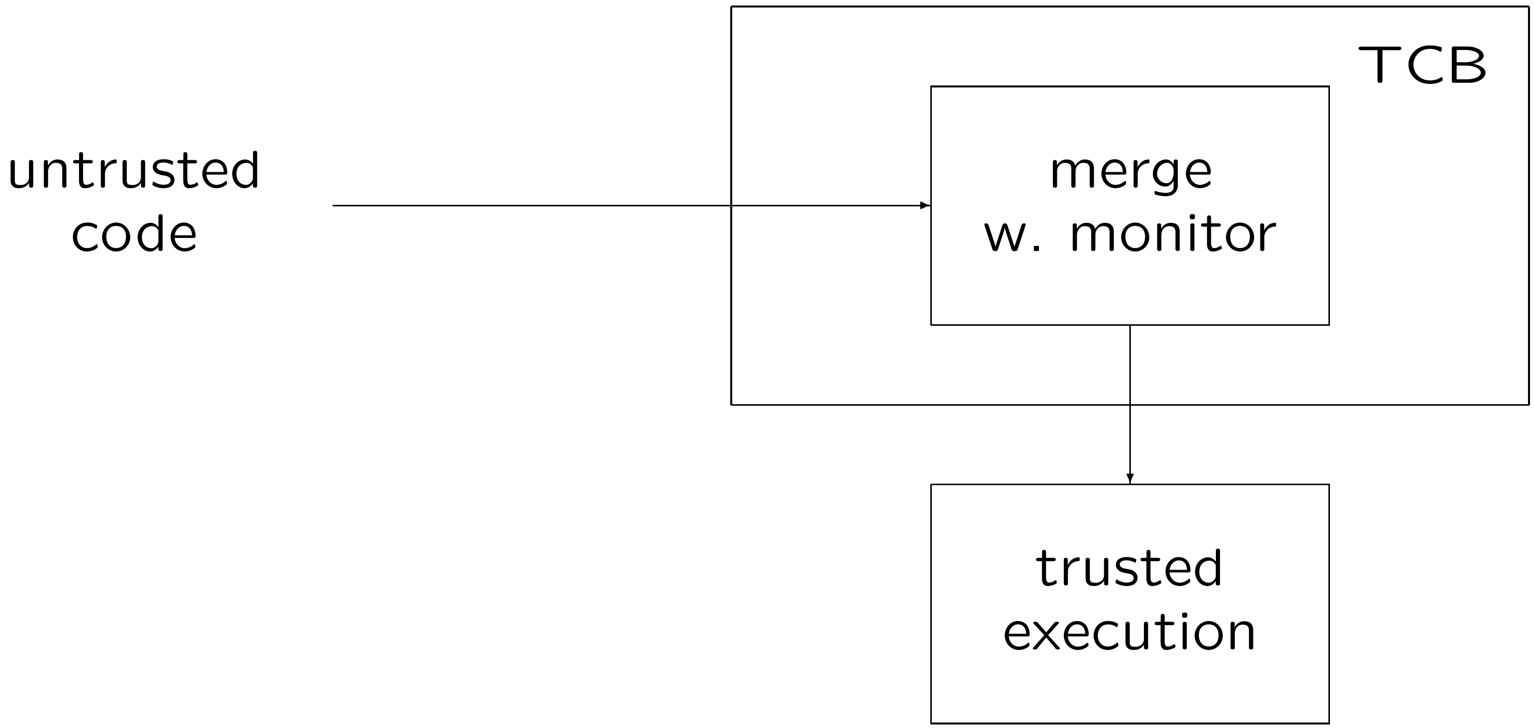
- *Memory safety:*
  dereference only valid pointers,
  memory access allowed and aligned.
- *Control-flow safety:*
  jump only to valid and allowed addresses.
- *Type safety:*
  program operations only on values of appropriate type.
- Type safety subsumes memory and control-flow safety.
- Many other possibilities.

Reference Monitor



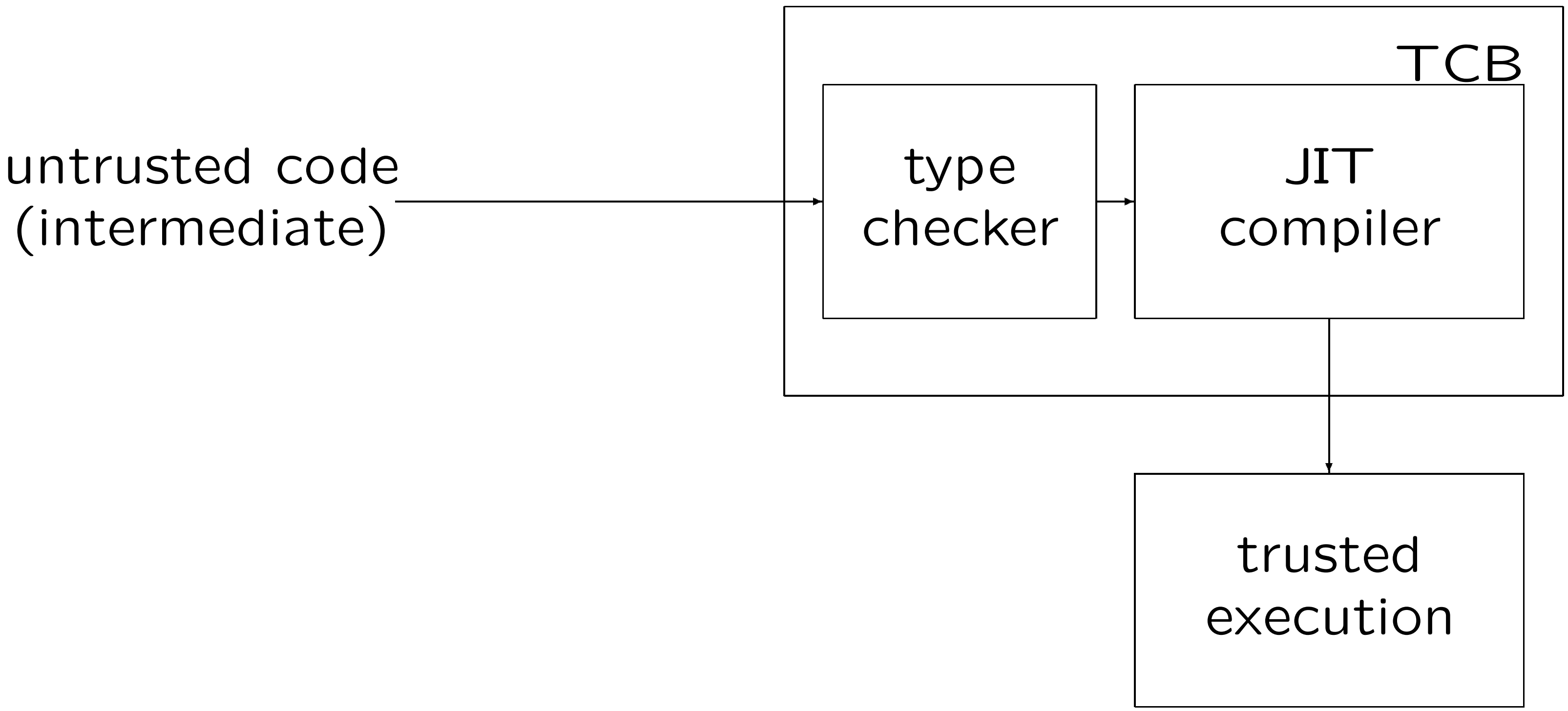
- Monitor (software or hardware) aborts unsafe execution.
- Burden on code consumer, inefficient.
- Difficult to enforce high-level abstractions.

Software Fault Isolation (SFI)



- Allows different languages and sources.
- Burden on code consumer, somewhat inefficient.
- Difficult to enforce high-level abstractions.

Just-in-Time Compiler

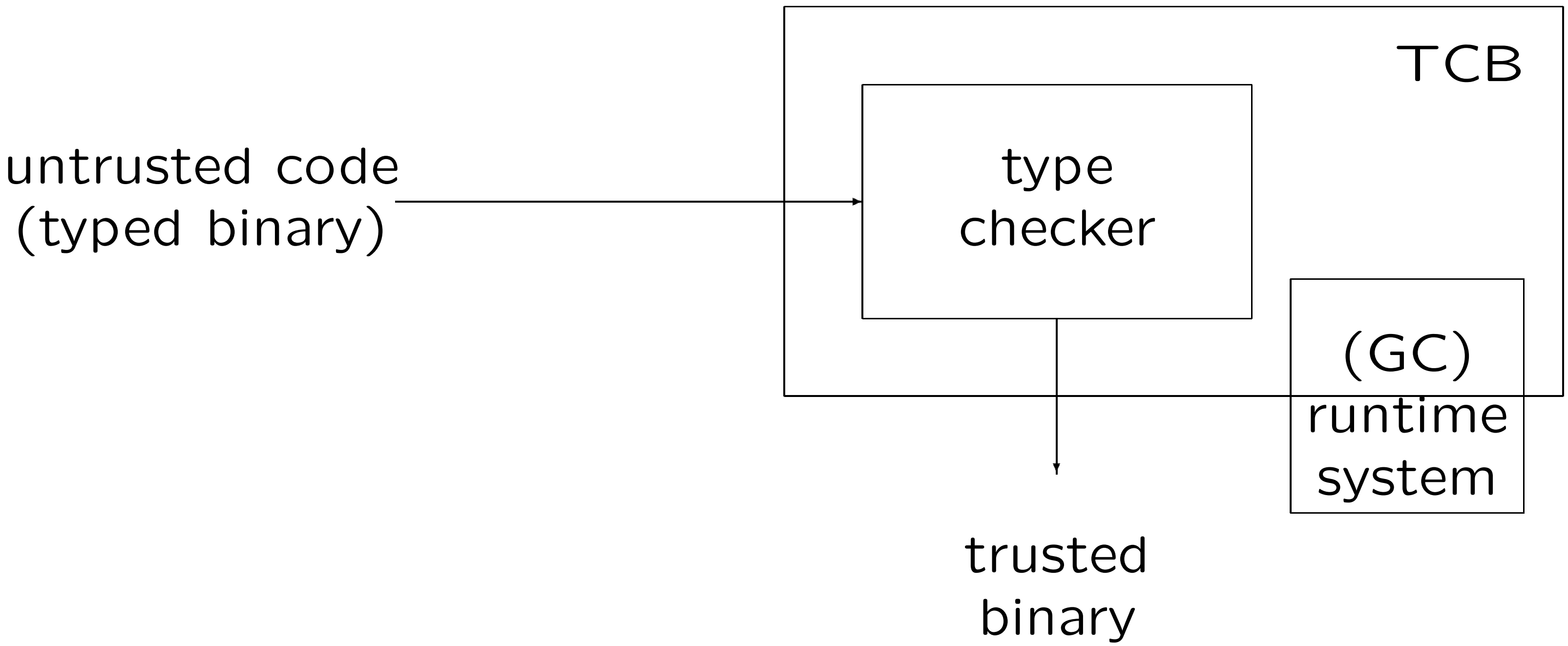


- Large, complex trusted computing base.
- Efficient execution.

Typed Assembly Language (TAL)

- Types as a syntactic discipline for enforcing levels of abstraction.
- Works well for high-level languages.
- Why not for assembly language or binaries?
  [Morrisett, Walker, Crary, Glew'98]

TAL Safety Architecture



- Small trusted computing base.
- Some overhead, some restrictions.

Questions about TAL

- What does the type system look like?
- How do we obtain a typed binary?
- How to we prove soundness?
  (well-typed programs cannot go wrong)
- Overhead in space and time?
- Restrictions on the form of code?

Example — TAL Type System

- Function computing factorial of `r1`, returning to `r2`.

```
fact:
  code{r1:int,r2:{r1:int}}.
    mov r3,1            set up accumulator for loop
    jmp loop
loop:
  code{r1:int,r2:{r1:int},r3:int}.
    bz r1,done          check if done, branch if zero
    mul r3,r3,r1
    sub r1,r1,1
    jmp loop
done:
  code{r1:int,r2:{r1:int},r3:int}.
    mov r1,r3           move accumulator to result register
    jmp r2              return to caller
```

Typed Intermediate Languages

- Start with a safe source language.
- Maintain type information throughout compilation.
- Annotate binary with types that cannot be readily inferred.
- Space overhead acceptable.
- Note: software fault isolation has no annotations to exploit.
- Burden is on the code producer.

TAL Discussion

- Easy to accomodate high-level invariants.
- Low-level type system tailored to source type system.
- Can interfere with optimizations.
- Type system engineered for a specific safety policy.
- Mathematical soundness proofs not easy.
- Tampering does not impact safety.
- Caveat: guarantees only as strong as the mathematical model of the machine.
  (example: separation of program and data)

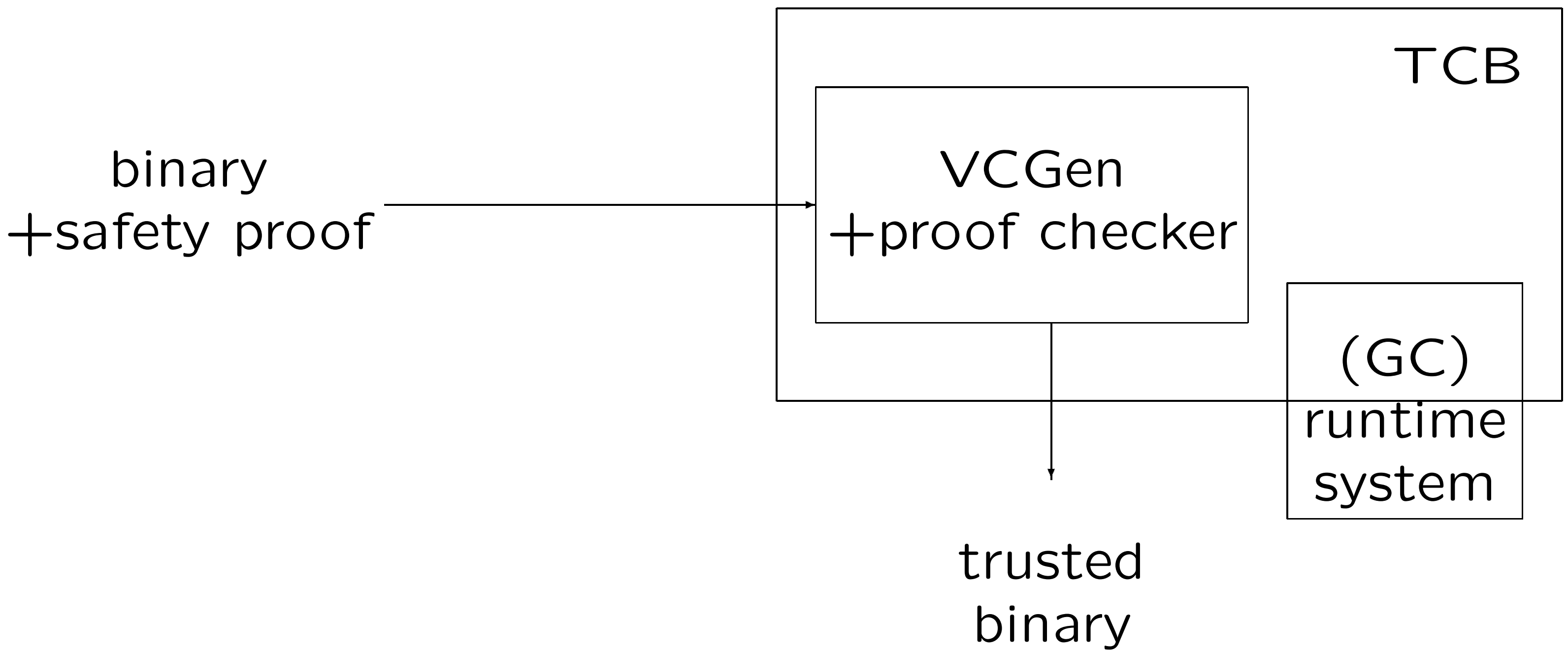
TAL State-of-the-Art

- Original TAL for "safe C". [Morrisett, Walker, Crary, Glew'98'99]
- TILT — ML types to RTL level.
  [Morrisett'95] [Morrisett, Harper, et al.'96]
- TAL with resource bounds. [Crary & Weirich'00]
- DTAL — dependently typed assembly language.
  Stronger invariants for efficiency and increased reliability.
  [Xi& Pf'98][Xi& Harper'99]

Proof-Carrying Code (PCC)

- Code producer attaches a proof that binary is safe.
- Code consumer checks the proof against the code.
- Then discards the proof, runs the binary.

PCC Safety Architecture



- VCGen = *Verification Condition Generator*
- Small trusted computing base.
- Need small, efficiently checkable proof objects.

Questions about PCC

- What do safety proofs look like?
- How do we obtain a safety proof?
- How do we prove soundness
  (provably safe programs are really safe)?
- Overhead in space and time?
- Restrictions on the form of code?

Formal Safety Policies

- Safety policy given by inference rules.
- Generic rules for logical propositions.
- Specific rules for safety propositions (saferead(a), safewrite(a), int(r1), ...)

$$\frac{a \bmod 4 = 0 \qquad \text{accessible}(a)}{\text{saferead}(a)}$$

Proof Objects in Logical Framework (LF)

$$\frac{A \quad B}{A \wedge B}\wedge I \qquad \frac{A \wedge B}{A}\wedge E_1 \qquad \frac{A \wedge B}{B}\wedge E_2 \qquad \frac{\begin{array}{c}\overline{A}^{\,u}\\ \vdots \\ B\end{array}}{A \supset B}\supset I^u$$

- Inference rules as functions from proofs of premises to proofs of conclusion.

```
andi : pf A -> pf B -> pf (A & B).
ande1 : pf (A & B) -> pf A.
ande2 : pf (A & B) -> pf B.
impi : (pf A -> pf B) -> pf (A => B).
impi(λu. andi (ande1 u) (ande2 u)) : pf (A & B => B & A).
```

LF Representation

- Logical framework: a meta-language for specifying logics and representing proofs.
- Safety policy specified as signature.
  (list of constant declarations)
- Proof-checking is type-checking!
- LF contains many redundancies.
- Syntactic redundancies can be eliminated.
  [Michaylov & Pf'93] [Necula'98] [Pf & Schürmann'98]
- Proof-checking quite efficient in practice.

Certifying Compilation

- Start with a safe source language.
- Maintain invariants throughout compilation.
- Apply the verification condition generator (VCGen).
  (requires invariants)
- Prove the verification condition.
  (should be provable if compiler is correct)
- Use cooperating decision procedures.

Example: Safe Array Access

```
if (0 <= i && i <= *A) {
   return A[i+1] /* unsafe access */
} else {
   ... signal an error ...
}
```

- Safe implementation of array access `sub(A,i)`.
- Integer array as pointer to a sequence of words.
- First contains array's length.
- Next: annotate with assertions.

Example: Adding Logical Assertions

```
/* int i, array A */
if (0 <= i && i <= *A) {
   /* 0 <= i < length(A) */
   return A[i+1]
} else {
   ... signal an error ...
}
```

- Invariants from source-level declarations.
- Invariants from control flow.
- Next: use `sub(A,i)` in array summation.

Example: Summing an Array

```
int sum = 0;
for (i=0; i<length(A); i++) {
  /* 0 <= i < length(A) */
  sum += sub(A,i);  /* safe access */
}
```

- Propagate assertion through code for `sub`.
- Next: in-line and optimize.

Example: Assertion-Based Optimization

```
int sum = 0;
for (i=0; i<*A; i++) {
  /* 0 <= i < length(A) */
  sum += A[i+1];    /* unsafe access, proven safe */
}
```

- Unfold (inline) definition of `sub`.
- Eliminate bounds check.
- Next: annotate with proof of assertion.

Example: Certified Intermediate Code

```
int sum = 0;
for (i=0; i<*A; i++) {
  /* π : 0 <= i < length(A) */
  sum += A[i+1]
}
```

- Similar at machine code level.

Soundness

- Rigorous mathematical proof. [Necula'98]
- Partial formalization in linear logical framework. [Plesko & Pf'99]
- Building a theory of types from machine model. [Appel & Felty'00]
- Correctness of signature in practice?

PCC Discussion

- Space overhead highly variable.
- Run-time overhead manageable.
- Compile-time overhead manageable.
- Code efficiency comparable or better than standard compilers.
- Most burden on code producer.
- More flexible than TAL.
- Less systematic than TAL.

PCC State-of-the-Art

- Original Touchstone compiler for "safe C". [Necula'98]
- Original proof sizes 2x to 4x of binary.
- Special J certifying compiler for Java [Colby, Lee, Necula et al.'00]
- Certifies memory, control, and type safety.
- Compiles 300 real-word Java applications, including Hotjava (150K lines), StarOffice (100K lines).
- Annotations and proofs 25%–40% of machine code.
- Proofs represented as "oracle strings".

The Real Lesson

- **Type theory** and **logic** are indispensible for solving system problems!
- All compilers and theorem provers should be certifying!
- Valuable development and debugging tool.
  (Twelf, Touchstone, Special J, CASC)
- Increased confidence **and** increased efficiency.

Future Work

- TILT Compiler for ML to TAL.
- Stronger invariants for both TAL and PCC (refinement types and dependent types).
- Formally verifying soundness using meta-logical framework (PCC signatures, TAL type systems)
- Proof compression and analysis.