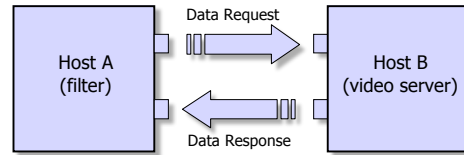


# Proof-Carrying Code and Typed Assembly Language

CMSC 631  
Fall 2006

Adapted from slides by David Walker  
for Princeton course CoS 598E

## Today's Systems

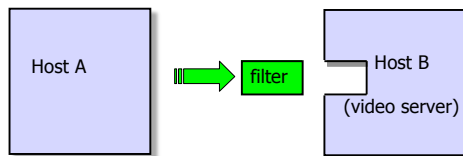


### Static, narrow interfaces & protocols

- data must be copied to/from hosts
- new functionality is performed at the client
- bandwidth wasted when server cycles are cheap

2

## Tomorrow's Systems: Agents



A sends a *computational agent* to B

- Agent encapsulates data & code
- Agent runs locally with direct access to data
- Agent returns to A with results

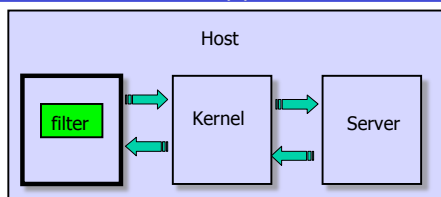
3

## Some Critical Issues:

- Safety concerns:
  - how to protect from faulty/malicious agent?
- Complexity concerns:
  - how to minimize trusted computing base?
- Performance concerns:
  - how to minimize overheads?
- Other concerns (not addressed here):
  - how to ensure privacy and authenticity?
  - How to protect agent from host?

4

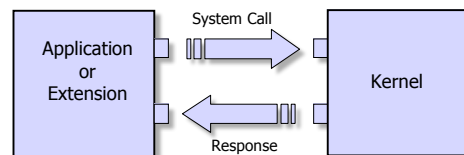
## Traditional OS-Approach



Place extension in separate address space  
and re-direct system calls.

5

## It's the same problem...



And many of the same issues!

- Static, narrow interface
- No data sharing (must copy to/from Kernel)
- Overhead of context switch, TLB-flush, etc.

6

## Everyone wants extensibility:

- OS Kernel
- Web browser
- Routers, switches, "active" networks
- Servers, repositories

All face the same problem:  
How to give agents direct access  
without compromising host integrity?

7

## Idea for Solution: Type-Safety

Agent is written in a "type-safe" language:

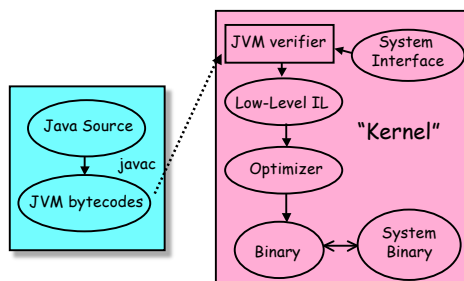
- type safe ~ respects host's abstractions
- Java, Modula-3, ML, Scheme

Host ensures agent code is legal:

- static checks (e.g., type-checking)
- dynamic checks (e.g., array-bound checks)
  - requires interpreter or JIT compiler

8

## Example: JVM



9

## JVM Pros & Cons

Pros:

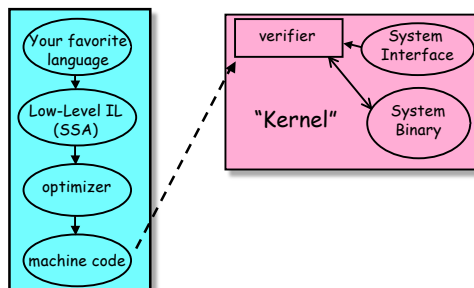
- portable
- hype: \$, tools, libraries, books, training

Cons:

- large trusted computing base
  - includes JVM interpreter/JIT
- requires many run-time tests
  - "down" casts, arrays, null pointers, etc.
- only suitable for Java (too high-level)
  - even then, there are mismatches with JVM
- no formal spec (yet)

10

## Ideally:



11

## Proof Carrying Code [Necula & Lee]

Fundamental idea:

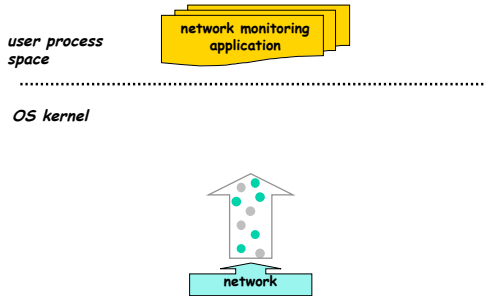
- finding a proof is hard
- verifying a proof is easy
- (not so apparent to systems people)

PCC:

- agent = highly optimized code + proof that code respects the host's integrity.

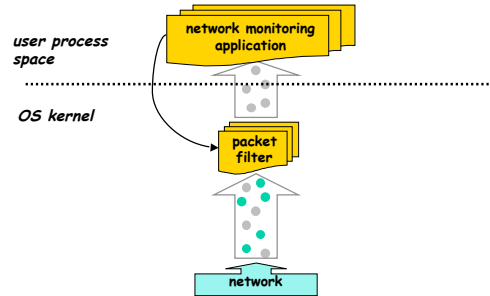
12

## Example: Packet Filters



13

## Example: Packet Filters



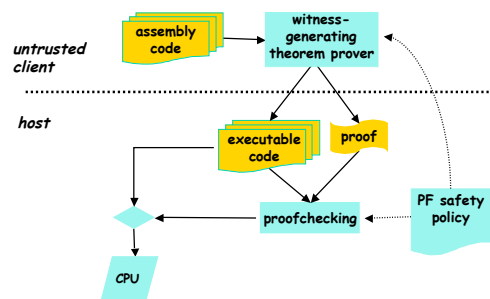
14

## An Experiment:

- Safety Policy:
  - given a packet, returns yes/no
  - packet read only, small scratchpad
  - no loops
- Experiment: [Necula & Lee, OSDI'96]
  - Berkeley Packet Filter Interpreter
  - Modula-3 (SPIN)
  - Software Fault Isolation (sandboxing)
  - PCC

15

## Packet Filters in PCC



16

## Packet Filter Summary

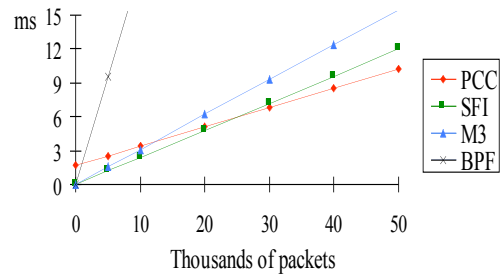
The PCC packet filter worked extremely well:

- BPF safety policy was easy to verify automatically.
  - r0 is aligned address of network packet (read only)
  - r1 is length of packet ( $\geq 64$  bytes)
  - r2 is aligned address of writeable 16-byte array
- Allowed hand-optimized packet filters.
  - The "world's fastest packet filters".
  - 10 times faster than BPF.
- Proof sizes and checking times were small.
  - About 1ms proof checking time.
  - 100%-300% overhead for attached proof.

17

## Results:

PCC wins:



18

## Advantages of PCC

- Simple, small, and fast trusted computing base.
- No external authentication or cryptography.
- No need to insert additional run-time checks.
- "Tamper-proof".
- Precise and expressive specification of code safety policies.

19

## One Approach to PCC:

- Axiomatic semantics
- Safety policy: additional pre-conditions
- Floyd/Hoare-style verification condition generator (VCgen) converts to first-order predicate
  - loop-invariants needed as annotations
  - proof of VC implies code respects safety policy

20

## Challenges with PCC

- What to put in the safety policy?
  - Which safety properties?
  - Liveness properties?
- How to automate proof construction?
  - How to automate loop invariants?
- Engineering issues:
  - complexity of proof checker?
    - This is where Andrew's group & FPCC comes in...
  - size and time to check proofs?

21

## Proof Representation

PCC uses the Edinburgh Logical Framework

- LF type checking adequate for proof checking.
- Flexible framework for encoding security policies (i.e., logics).
- Type-checker is small (~6 pages of C code)

22

## Certifying Compilers

Big question is still: how to get proofs?

Focus on traditional type-safety as the security policy.

Source is well-typed... just need to maintain the proof throughout compilation...

23

## Type-Preserving Compilers

TIL compiler [PLDI'96]

high-performance ML  
types only to intermediate level

Touchstone [PLDI'98]

simple, low-level language  
emits high-performance PCC

TALC [PoPL'98, TIC'98]

general type system for assembly language(s)  
techniques for compiling high-level languages

24

## Compiling High-Level Languages

When encoding constructs, we must:

- preserve typing (to emit invariants)
- preserve abstraction
  - users reason about security at the source level.
  - security policy may be a source-level interface (e.g., an ML signature).
  - cannot afford for agent machine code to violate these abstractions.
- somehow get to the machine level

25

## TAL [PoPL'98, TIC'98]

Type system for assembly language.

- and a set of encodings for high-level language features.
- certifying compilers for:
  - Safe-C, Scheme, ML

Type system based on System-F

- operational model (accurate machine)
- "syntactic" soundness result

26

## TALx86

Type-checker and tools for Intel x86 code.

Slightly different framework than PCC:

- don't ship explicit proof
  - they're very large due to encodings.
- instead, ship code with invariant annotations
- host reconstructs & verifies proof
  - + smaller agents, potentially faster checking
  - larger TCB, less flexible framework

27

## TAL Built-In Types

- Bytes: int1, int2, int4, ..
- Tuples:  $(\tau_1^{\phi_1}, \dots, \tau_n^{\phi_n})$  ( $\phi = 0, 1$ )
- Code:  $\{r_1:\tau_1, \dots, r_n:\tau_n\}$ 
  - think pre-condition
- Polymorphic types:  $\forall\alpha.\tau, \exists\alpha.\tau$

28

## Simple Loop

```
sum: {ecx:int4, ebx:{eax:int4}}           ; int sum(int x) {
  mov  eax,0                               ; int a = 0;
  jmp  test                                 ;
loop: {eax:int4, ecx:int4, ebx:{eax:int4}}; while(!x) {
  add  eax,ecx                               ; a += x;
  dec  ecx                                   ; x--;
  FALLTHRU                                  ; }
test: {eax:int4, ecx:int4, ebx:{eax:int4}};
  cmp  ecx,0                               ;
  jne  loop                                 ; return(a);
  jmp  ebx                                   ; }
```

29

## Other TAL Features

- Module system
  - interfaces, implementations, ADTs
- Records and datatypes
- Arrays/vectors
- (Higher Order) Type constructors
- Stack support
- Aliasing support

30

## TAL-0: Control flow safety

- Goal: ensure that the program only jumps to one of a set of well-defined entry points
- TAL-0 syntax:
  - $r ::= r_1 \mid r_2 \mid \dots \mid r_k$
  - $v ::= n \mid l \mid r$
  - $i ::= r_d := v$ 
    - $\mid r_d := r_s + v$
    - $\mid \text{if } r \text{ jump } v$
  - $I ::= \text{jump } v \mid i; I$

31

## Example: compute $r3 := r1 * r2$

```

Prod: r3 := 0;           // res := 0
      jump loop

Loop: if r1 jump done;  // if a=0 goto done
      r3 := r2 + r3;    // res := res+b
      r1 := r1 + -1;    // a = a-1
      jump loop;

Done: jump r4           // return
  
```

32

## TAL-0 Abstract Machine

- $R ::= \{ r_1 = v_1, \dots, r_k = v_k \}$  registers
- $H ::= \{ l_1 = I_1, \dots, l_m = I_m \}$  heap
- $M ::= (H, R, I)$  machine state

- Define rewriting relation  $M \rightarrow M'$
- Define
  - $R(r) = R(r)$
  - $R(n) = n$
  - $R(l) = l$

33

## Operational Rules

$$\frac{H(R(v)) = I}{(H, R, \text{jump } v) \rightarrow (H, R, I)}$$

$$\frac{R(r_s) = n_1 \quad R(v) = n_2}{(H, R, r_d := r_s + v; I) \rightarrow (H, R[r_d = n_1 + n_2], I)}$$

$$\frac{R(r) = n \quad n \neq 0}{(H, \text{if } r \text{ jump } v; I) \rightarrow (H, R, I)}$$

$$\frac{R(r) = 0 \quad H(R(v)) = I'}{(H, \text{if } r \text{ jump } v; I) \rightarrow (H, R, I')}$$

$$(H, R, r_d := v; I) \rightarrow (H, R[r_d = R(v)], I)$$

34

## What can go wrong?

- The abstract machine gets stuck if
  - It tries to jump to an integer, rather than a label
  - It tries to add an integer to a label
  - It tries to test a label using if
- We can use a type system to prevent these things from happening
  - I.e., if a TAL-0 program is well typed, it will never get stuck

35

## TAL-0 Types Syntax

- Operand types
  - $T ::= \text{int} \mid \text{code}(A) \mid \alpha \mid \forall \alpha. T$
- Register file types
  - $A ::= \{ r_1: T_1, \dots, r_k: T_k \}$
- Heap types:
  - $\Psi ::= \{ l_1: T_1, \dots, l_n: T_n \}$

36

## TAL-0 Typing Rules

Values	$\Psi \vdash v : \tau$	Instruction Sequences	$\Psi \vdash I : \tau$
$\Psi \vdash n : \text{int}$	(S-INT)	$\frac{\Psi; \Gamma \vdash v : \text{code}(\Gamma)}{\Psi \vdash \text{jump } v : \text{code}(\Gamma)}$	(S-JUMP)
$\Psi \vdash \ell : \Psi(\ell)$	(S-LAB)	$\frac{\Psi \vdash \ell : \Gamma \rightarrow \Gamma_2 \quad \Psi \vdash I : \text{code}(\Gamma_2)}{\Psi; \Gamma \vdash v : \tau}$	(S-SEQ)
<b>Operands</b>	$\Psi; \Gamma \vdash r : \Gamma(r)$	$\frac{\Psi \vdash \ell : \Gamma \rightarrow \Gamma_2 \quad \Psi \vdash I : \tau}{\Psi \vdash I : \tau}$	(S-GEN)
$\Psi \vdash v : \tau$	(S-REG)	$\frac{\Psi; \Psi \vdash R(r) : \Gamma(r)}{\Psi \vdash R : \Gamma}$	(S-REGFILE)
$\Psi; \Gamma \vdash v : \tau$	(S-VAL)	<b>Heaps</b>	$\frac{\Psi \vdash H : \Psi}{\vdash H : \Psi}$
$\frac{\Psi; \Gamma \vdash v : \forall \alpha. \tau}{\Psi; \Gamma \vdash v : \tau[\tau/\alpha]}$	(S-INST)	$\frac{\forall \ell \in \text{dom}(\Psi). \Psi \vdash H(\ell) : \Psi(\ell) \quad \text{FTV}(\Psi(\ell)) = \emptyset}{\vdash H : \Psi}$	(S-HEAP)
<b>Instructions</b>	$\frac{\Psi; \Gamma \vdash v : \tau}{\Psi \vdash r_d := v : \Gamma \rightarrow \Gamma[r_d : \tau]}$	(S-MOV)	$\frac{\Psi; \Gamma \vdash r_1 : \text{int} \quad \Psi; \Gamma \vdash v : \text{int}}{\Psi \vdash r_d := r_1 + v : \Gamma \rightarrow \Gamma[r_d : \text{int}]}$
$\frac{\Psi; \Gamma \vdash r_1 : \text{int} \quad \Psi; \Gamma \vdash v : \text{int}}{\Psi \vdash r_d := r_1 + v : \Gamma \rightarrow \Gamma[r_d : \text{int}]}$	(S-ADD)	<b>Machine States</b>	$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash I : \text{code}(\Gamma)}{\vdash (H, R, I)}$
$\frac{\Psi; \Gamma \vdash r_1 : \text{int} \quad \Psi; \Gamma \vdash v : \text{code}(\Gamma)}{\Psi \vdash \text{if } r_1 \text{ jump } v : \Gamma \rightarrow \Gamma}$	(S-IF)		

## Example typing

```
H(Loop) =
  if r1 jump Done;
  r3 := r2 + r3;
  r1 := r1 + -1;
  jump Loop;
```

let  $\Gamma = \{r1, r2, r3 : \text{int}, r4 : \forall \alpha. \text{code}\{r1, r2, r3 : \text{int}, r4 : \alpha\}\}$

let  $\psi = \{\text{Prod} : \Gamma; \text{Loop} : \Gamma; \text{Done} : \Gamma\}$

Prove  $\vdash\!-\! H : \psi$

38

## Typing Instructions

$$\frac{\psi \vdash\!-\! \text{Done} : \psi(\text{Done}) = \text{code}(\Gamma) \quad \psi; \Gamma \vdash\!-\! r1 : \Gamma(r1) = \text{int} \quad \psi; \Gamma \vdash\!-\! \text{Done} : \text{code}(\Gamma)}{\psi \vdash\!-\! \text{if } r1 \text{ jump Done} : \Gamma \rightarrow \Gamma} \quad (1)$$

$$\frac{\psi; \Gamma \vdash\!-\! r2 : \Gamma(r2) = \text{int} \quad \psi; \Gamma \vdash\!-\! r3 : \Gamma(r3) = \text{int}}{\psi \vdash\!-\! r3 := r2 + r3 : \Gamma \rightarrow \Gamma}$$

$$\frac{\psi; \Gamma \vdash\!-\! r1 : \Gamma(r1) = \text{int} \quad \psi \vdash\!-\! -1 : \text{int}}{\psi \vdash\!-\! r1 := r1 + (-1) : \Gamma \rightarrow \Gamma}$$

$$\frac{\psi \vdash\!-\! \text{Loop} : \psi(\text{Loop}) = \text{code}(\Gamma) \quad \psi; \Gamma \vdash\!-\! \text{Loop} : \text{code}(\Gamma)}{\psi \vdash\!-\! \text{jump Loop} : \text{code}(\Gamma)}$$

39

## Stringing together the proofs

$$\frac{\text{string together other proofs here ...} \quad (1) \quad \psi \vdash\!-\! \text{if } r1 \text{ then jump Loop} : \Gamma \rightarrow \Gamma \quad \psi \vdash\!-\! I : \text{code}(\Gamma)}{\psi \vdash\!-\! \text{if } r1 \text{ then jump Loop} : I : \text{code}(\Gamma)}$$

- Question: how is polymorphism useful?
  - Where would you run into problems if you didn't have it?

40

## TAL-0 Soundness

- If  $\vdash\!-\! M$  then there exists  $M'$  such that  $M \rightarrow M'$  and  $\vdash\!-\! M'$ .
  - Proof by induction on the structure of  $M$ .

41

## TAL-1: Memory Safety

- So far, the heap may only contain code
- Real programs use pointers (to the heap) and dynamic memory allocation
- Approach:
  - Model the heap as also containing tuples of primitive values, add operations to load and store to the heap
  - Add primitives for performing heap and stack allocation, and heap deallocation
  - Define types that prove these operations are well-formed.

42

## Conclusions

---

- Ideas from high-level language type systems can be applied to low-level languages, too.
- Can form the foundation of techniques like proof-carrying code
  - More efficient than interpreters, domain crossings, etc.
- Can read more in the TAL papers, and in the TAL chapter in ATTAPL.

43