Proof-Carrying Code and Typed Assembly Language

CMSC 631
Fall 2006
Adapted from slides by David Walker for Princeton course Co5 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

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

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?

Traditional OS-Approach

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

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.
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?

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

Example: JVM

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)

Ideally:

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.
Example: Packet Filters

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

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 (=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.

Results:

PCC wins:
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.

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

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?

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)

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...

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
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

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

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

TAL Built-In Types

• Bytes: int1, int2, int4, ..
• Tuples: (τ₁^φ₁, .., τₙ^φₙ) (φ = 0,1)
• Code: {r₁:τ₁, .., rₙ:τₙ}
  • think pre-condition
• Polymorphic types: ∀α.τ, ∃α.τ

Simple Loop

```
sum: {ecx:int4, ebx:{eax:int4}} ; int sum(int x) {
  mov eax,0 ; int a = 0;
  jmp loop ;
loop: {eax:int4, ecx:int4, ebx:{eax:int4}}; while(!x) {
  add eax,ecx ; a += x;
  dec ecx ; x--;
  FALTHRU ;
} test: {eax:int4, ecx:int4, ebx:{eax:int4}};
  cmp ecx,0 ;
  jne loop ; return(a);
  jmp ebx ;
```
**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:**
  
  
  \[
  r ::= r_1 \mid r_2 \mid \ldots \mid r_k \\
  v ::= n \mid l \mid r \\
  i ::= r \\
  d ::= v \\
  s ::= v \\
  jump v \\
  i; I \\
  \]

**Example: compute** \( r_3 := r_1 \times r_2 \)

**Prod:** \( r_3 := 0; \) // res := 0

**jump loop**

**Loop:** if \( r_1 \) jump done; // if a=0 goto done

\( r_3 := r_2 + r_3; \) // res := res+b

\( r_1 := r_1 - 1; \) // a := a-1

**jump loop**

**Done:** jump \( r_4 \) // return

---

**TAL-0 Abstract Machine**

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

**Define rewriting relation** \( M \rightarrow M' \)

**Define**

- \( R(r) = R(r) \)
- \( R(n) = n \)
- \( R(l) = l \)

---

**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

---

**TAL-0 Types Syntax**

**Operand types**

- \( T ::= \text{int} \mid \text{code}(A) \mid \alpha \mid \forall \alpha . T \)

**Register file types**

- \( A ::= \{ r_1 : T_1, \ldots, r_k : T_k \} \)

**Heap types:**

- \( \Psi ::= \{ l_1 : T_1, \ldots, l_n : T_n \} \)
Example typing

\[ H(\text{Loop}) = \]
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 \( \vdash H : \psi \)

Stringing together the proofs

\[ \psi \vdash \text{if } r1 \text{ then jump } \text{Loop} : \Gamma \rightarrow \Gamma \]
\[ \psi \vdash \text{I} : \text{code}(\Gamma) \]

\[ (1) \]

string together other proofs here ...

\[ \psi \vdash \text{if } r1 \text{ then jump } \text{Loop} : \Gamma \rightarrow \Gamma \]
\[ \psi \vdash \text{I} : \text{code}(\Gamma) \]

\[ \psi \vdash \text{if } r1 \text{ then jump } \text{Loop} : \Gamma \rightarrow \Gamma \]
\[ \psi \vdash \text{I} : \text{code}(\Gamma) \]

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

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
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.