TALx86: A Realistic Typed Assembly Language

Dan Grossman, Fred Smith
Cornell University

Joint work with: Greg Morrisett,
Karl Crary (CMU), Neal Glew, Richard Samuels,
Dave Walker, Stephanie Weirich, Steve Zdancewic

Everyone wants extensibility:

- Web browser
  - applets, plug-ins
- OS Kernel
  - packet filters, device drivers
- "Active" networks
  - service routines
- Databases
  - extensible ADTs

The Language Approach

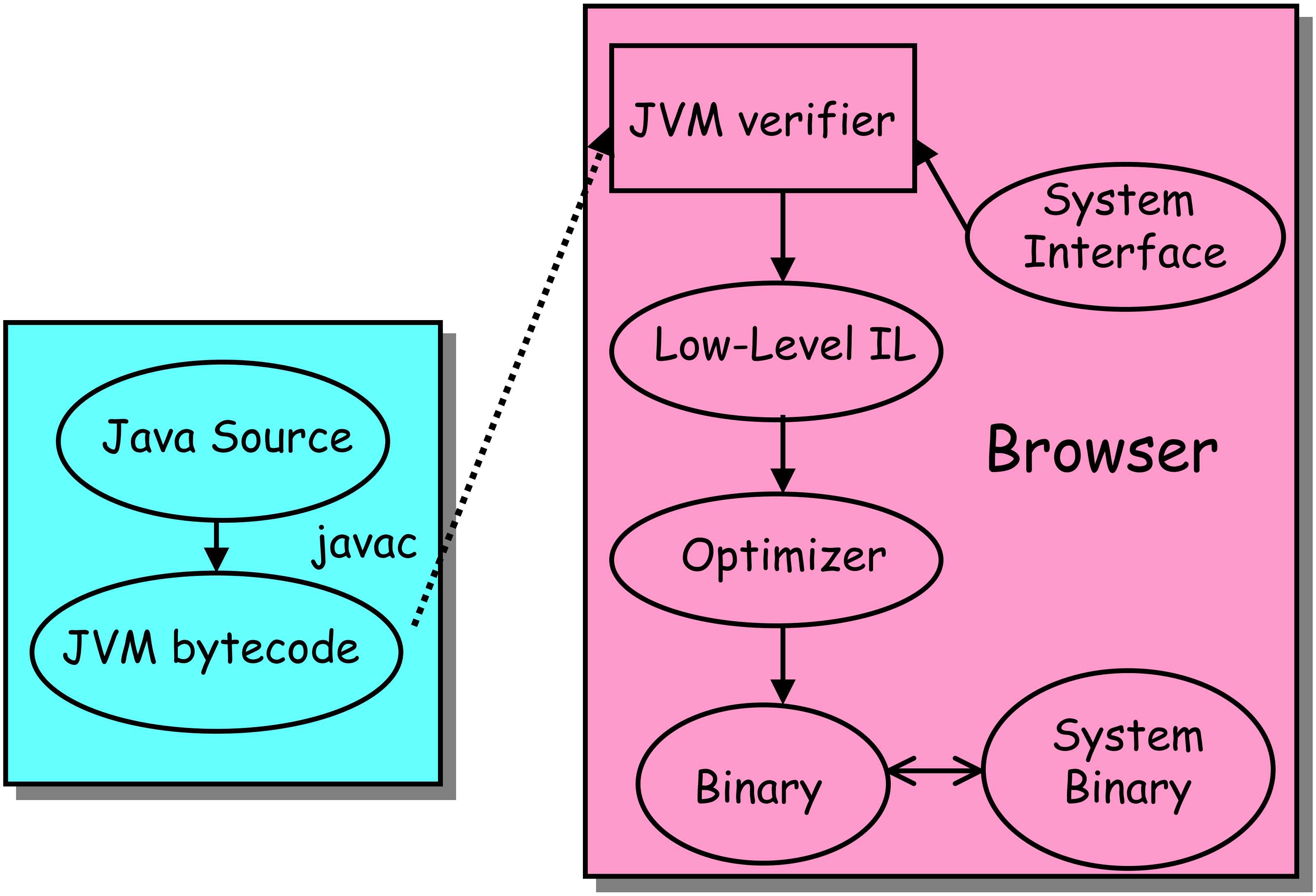
Extension is written in a "safe" language:

- Java, Modula-3, ML, Scheme
- Key point: language provides abstractions
  - ADTs, closures, objects, modules, etc.
  - Can be used to build fine-grained capabilities

Host ensures code respects abstractions:

- Static checking (verification)
- Inserting dynamic checks (code-rewriting)

Example: Your Web Browser



JVM Pros & Cons

Pros:

- Portability
- Hype: $, tools, libraries, books, training

Cons:

- Performance
  - unsatisfying, even with off-line compilation
- Only really suitable for Java (or slight variants):
  - relatively high-level instructions tailored to Java
  - type system is Java specific
- and...

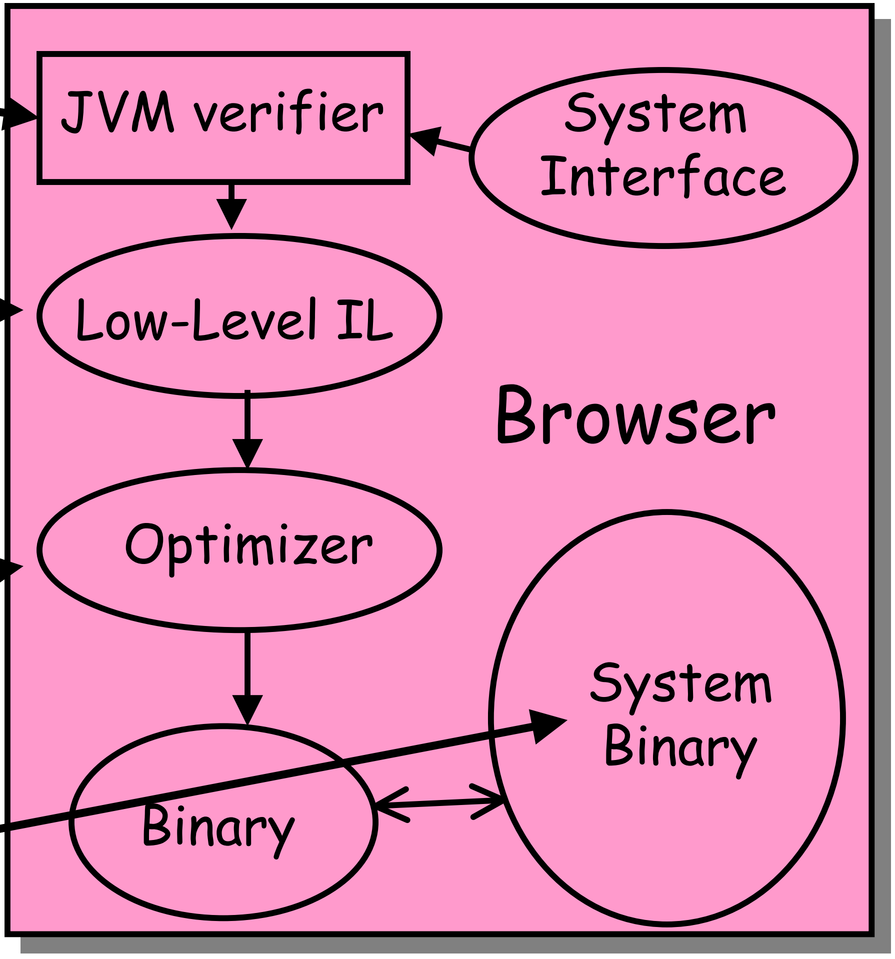
Large Trusted Computing Base

No complete formal model

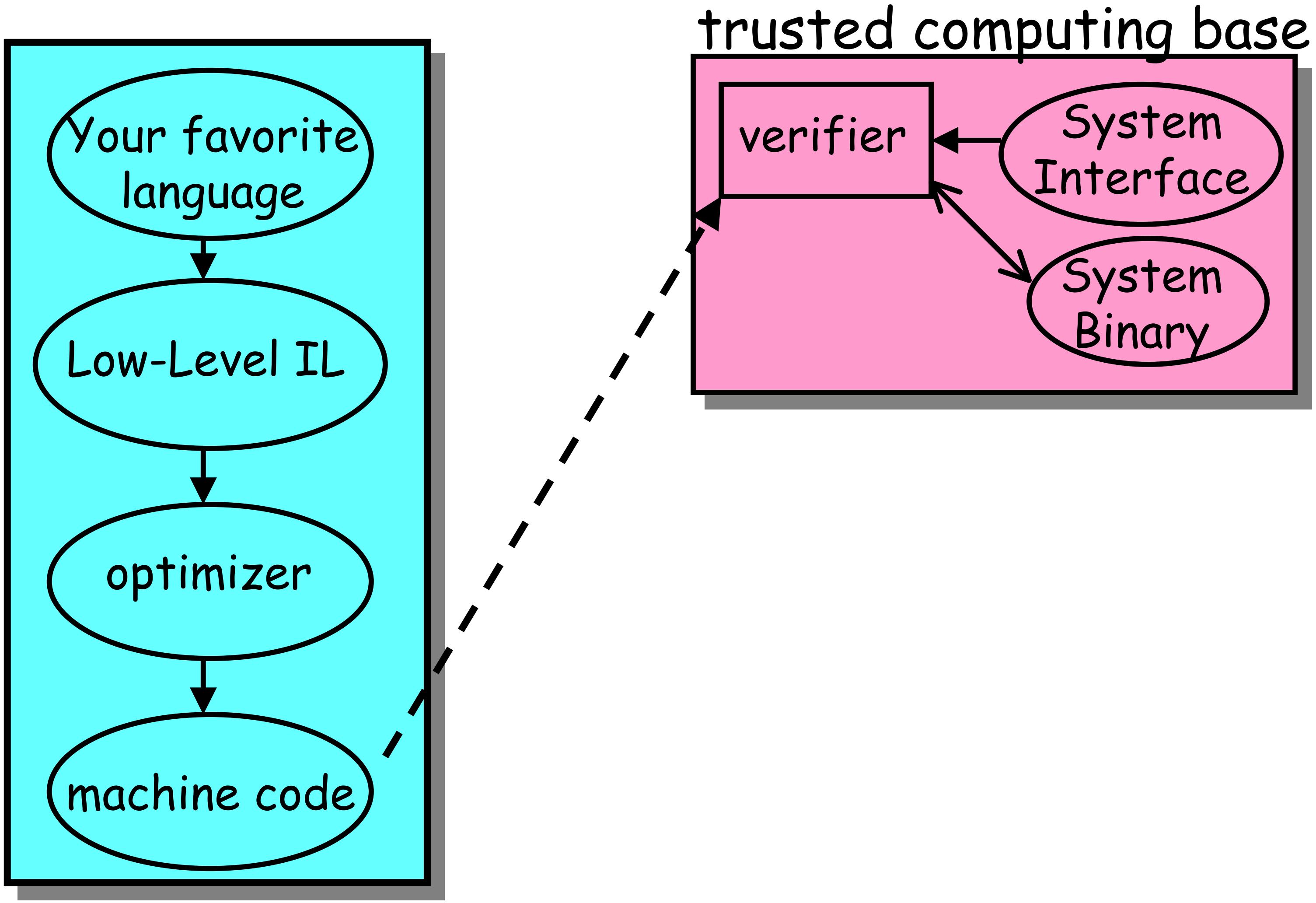
Must insert right checks

Good code ⇒ big optimizer ⇒ bugs in optimizer

Lots of "native" methods (i.e., not-safe code)

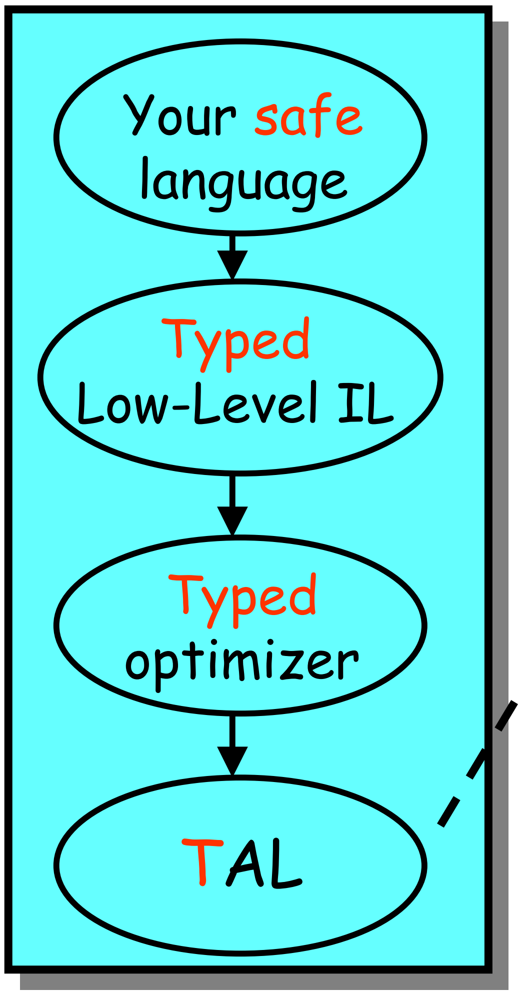


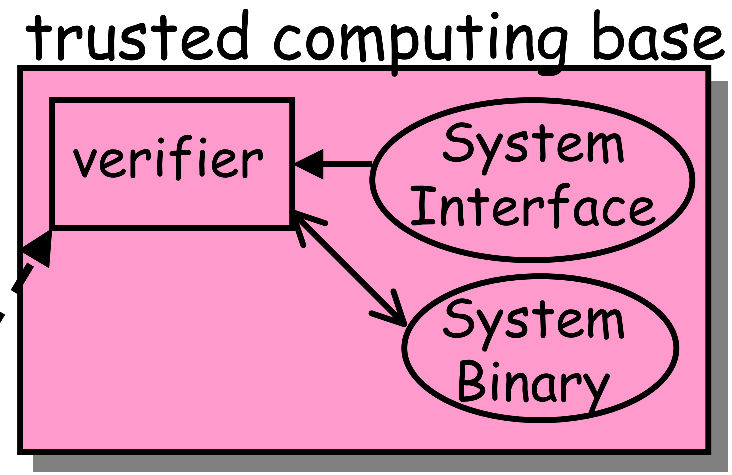
Ideally:



Your favorite language
Low-Level IL
optimizer
machine code
trusted computing base
verifier
System Interface
System Binary

The Types Way





- Verifier is a type-checker
- Type system flexible and expressive
- A useful instantiation of the "proof carrying code" framework

TALx86 in a Nutshell

- Most of the IA32 80x86 flat model assembly language
- Memory management primitives
- Sound type system

- Types for code, stacks, structs
- Other advanced features
- Future work (what we can't do yet)

TALx86 Basics:

Primitive types: (e.g., int)

Code types: $\{r_1:\tau_1,\ldots,r_n:\tau_n\}$

- *"I'm code that requires register* r_i *to have type* τ_i *before you can jump to me."*

- Code blocks are annotated with their types
  - Think pre-condition
  - Verify block assuming pre-condition

Sample Loop

C:

```
int sum(int n){
  int s=0;
  while(!n) {
    s+=n;
    --n;
  }
  return n;
}
```

TAL sketch:

<n and retn addr as input>

`sum:` ***<type>***

  <initialize s>

`loop:` ***<type>***

  <add to s, decrement n>

`test:` ***<type>***

  <return if n is 0>

Verification

```
sum:  {ecx:int, ebx:{edx:int}}
      mov   eax,0      {ecx:int, ebx:{edx:int}, eax:int}
      jmp   test       OK: sub-type of type labeling test
loop: {ecx:int, ebx:{edx:int}, eax:int}
      add   eax,ecx    {ecx:int, ebx:{edx:int}, eax:int}
      dec   ecx        {ecx:int, ebx:{edx:int}, eax:int}
        OK: sub-type of type labeling next block
test: {ecx:int, ebx:{edx:int}, eax:int}
      cmp   ecx,0      {ecx:int, ebx:{edx:int}, eax:int}
      jne   loop       OK: sub-type of type labeling loop
      mov   edx,eax    {ecx:int, ebx:{edx:int}, eax:int, edx:int}
      jmp   ebx        OK: sub-type of {edx:int} -- type of ebx
```

Stacks & Procedures

Stack Types (lists):

$$\sigma ::= \text{nil} \mid \tau::\sigma \mid \rho$$

where ρ is a stack type variable.

Examples using C calling convention:

`int square(int);`

$\forall\rho_1$ {esp: τ_1::int::ρ_1}

where

τ_1={eax: int, esp: int::ρ_1}

`int mult(int,int);`

$\forall\rho_2$ {esp: τ_2::int::int::ρ_2}

where

τ_2={eax: int, esp: int::int::ρ_2}

Stacks & Verification

square: $\forall\rho_1\,\{esp\!:\tau_1::int::\rho_1\}$

$\quad$ where $\tau_1=\{eax\!: int,\ esp\!: int::\rho_1\}$

```
    push [esp+4]
    push [esp+8]
    call mult
```
[with $\rho_2 = \tau_1::int::\rho_1$]

$\tau_{aft}=\{eax\!:int,\ esp\!: int::int::\tau_1::int::\rho_1\}$

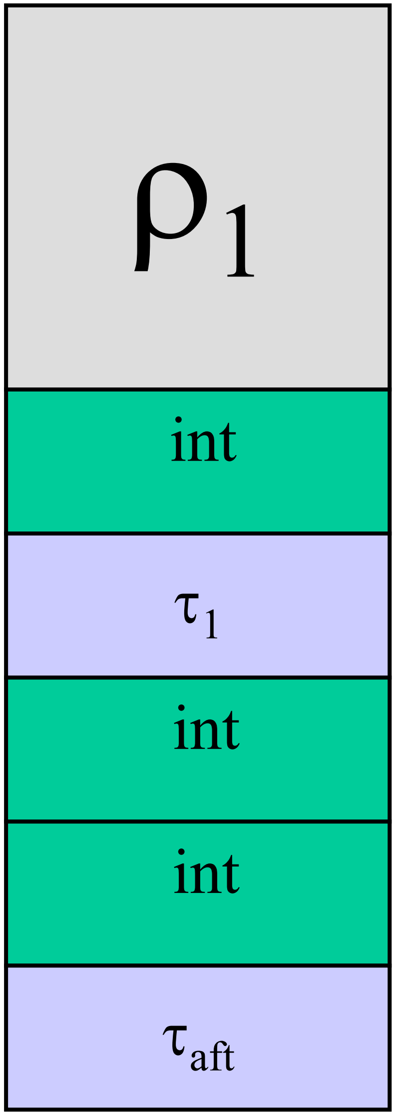
```
    add  esp,8
    retn
```

mult: $\forall\rho_2\,\{esp\!: \tau_2::int::int::\rho_2\}$

$\quad$ where $\tau_2=\{eax\!: int,\ esp\!: int::int::\rho_2\}$

$\{esp\!: \tau_2::int::int::\tau_1::int::\rho_1\}$

where $\tau_2=\{eax\!: int,\ esp\!: int::int::\tau_1::int::\rho_1\}$



Important Properties

- Abstraction

"Because the type of the rest of the stack is abstract the callee cannot read/write this portion of the stack"

- Flexibility

Can encode and enforce many calling conventions
(stack shape on return, callee-save, tail calls, etc.)

Callee-Save Example

mult:

$\forall\alpha \;\; \forall\rho_2$ {ebp: α, esp: τ_2::int::int::ρ_2}

where τ_2={ebp: α, eax: int, esp: int::int::ρ_2}

Structs

- Goals:
  - Prevent reading uninitialized fields
  - Permit flexible scheduling of initialization
- MALLOC "instruction"

  returns uninitialized record
- Type of struct tracks initialization of fields
- Example: {ecx: int}

```
MALLOC    eax,8 [int,int]   ; eax : ^*[int^u, int^u]
mov [eax+0], ecx            ; eax : ^*[int^rw,int^u]
mov ecx, [eax+4]            ; type error!
```

Much, much more

- Arrays (see next slide)
- Tagged Unions
- Displays, Exceptions [TIC'98]
- Static Data
- Modules and Interfaces [POPL'99]

- Run-time code generation
  [PEPM'99 Jim, Hornof]

Mis-features

- MALLOC and garbage collection in trusted computing base [POPL'99]
- No way to express aliasing
- No array bounds check elimination [Walker]
- Object/class support too primitive [Glew]

Summary and Conclusions

- We can type real machine code

  Potential for

  performance + flexibility + safety

- Challenge:

  Finding generally useful abstractions

- Lots of work remains

http://www.cs.cornell.edu/talc